/THEORY/IN/PRACTICE

你不可不知的关系数据库理论

Relational Theory for Computer Professionals

[美] C. J. Date 著

张大华 方帅 译

柳永坡 张奎 审

O'REILLY®



人民邮电出版社

POSTS & TELECOM PRESS

O'REILLY®

你不可不知的关系数据库理论

[美] C. J. Date 著
张大华 方 帅 译
柳永坡 张 奎 审

人 民 邮 电 出 版 社
北 京

版 权 声 明

内容提要

关系型数据库是建立在关系模型基础上的数据库，借助于集合代数等数学概念和方法来处理数据库中的数据。关系型数据库的基础——关系理论被认为是 SQL 的基础。

本书为我们讲解了什么才是真正的关系型数据库，与当前的数据库产品相比，它的特点和优势是什么。本书分为 3 部分，共计 14 章。第一部分是数据库的基础，讲解了数据库基本概念，关系和关系变量，码、外码和相关概念，关系运算符，约束和断言，关系模型等内容；第二部分讲解了事务的相关概念，以及如何设计一个好的数据库；第三部分则讲解了 SQL 相关的知识，其内容涵盖了 SQL 基本表，SQL 操作符和运算符，SQL 约束，SQL 与关系模型等内容。本书最后的 5 个附录涵盖了 Tutorial D 语法、TABLE_DUM 和 TABLE_DEE、集合论、关系演算，以及与关系理论知识相关的资源。

本书适合数据库开发、维护人员以及高校数据库专业的师生阅读。对于想要真正理解什么是关系型系统的读者来说，本书也是不错的选择。

O’Reilly Media, Inc.介绍

O’Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O’Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序

目前，现有大量新型数据库技术得到了快速发展并受到大量关注，无论是已有进展的“云计算”产业，还是流行的关注点“大数据”，其核心都是对“数据”的研究，这就使得数据库中的关系理论和 SQL 得到更进一步的丰富和发展。市面上关于 SQL 语言或者关系理论或者二者兼而有之的书籍参差不齐，良莠混杂，很难有一本能循序渐进深入讲解该领域的书籍。本书译者结合自己多年工作经验，结合实际行业内需求，挑选 C. J. Date 的这本图书供读者参考。

C. J. Date 是最早认识到 Code 在关系模型方面所做的开创性贡献的学者之一，他是关系数据库技术领域中非常著名的独立撰稿人、学者和顾问，他使得关系模型的概念普及化。他参与了 IBM 公司的 SQL/DS 和 DB2 两大产品的技术规划和设计。30 多年来，Date 一直活跃在数据库领域中，他的著作包括：《SQL 和关系理论（第二版）》、《数据库设计和关系理论》、《视图修改和关系理论》、《数据库系统导论》、《对象关系数据库基础：第三次宣言》等。

本书共分为 14 个章节，内容分别为关系型数据库“基础介绍”、“事务和数据库设计”、“SQL”及附录四个部分，翻译团队由中国电力科学研究院信息通信研究所相关人员组成。具体分工为：前言、译者序、“基础介绍”部分由张大华、方帅负责；“事务和数据库设计”由张大华、方帅、纪鑫、李哲、陈相舟负责；“SQL”部分由谢迎军、丁辉、常亮、刁倩、刘月林、钱声攀、魏俊负责；附录部分由上述全体翻译团队共同负责；全书译文审核与校正由北京航空航天大学柳永坡、张奎完成。

本书中文版能够出版发行，首先要感谢本书的作者，是他们为我们著作了一本好书。其次要感谢人民邮电出版社引进了本书，使我们获得了翻译此书的机会，并实现将其介绍给国内广大读者的良好愿望。也借此机会向本书的技术审校人员致以崇高的敬意，他们使本书的质量与水平向前迈进了一大步。此外，特别要感谢国家电网公司高级顾问曾楠，他以巨大的热情和高度的责任心在本书翻译过程中给予了大力的技术指导工作。也要感谢幕后的工作人员，是他们的辛勤劳动使得本书顺利付梓。在此，译者代表对所有为此书的出版做出贡献的人表示深深的感谢和崇高的敬意！

本书对原著的错误之处作了一些修正，在原著难懂或需要提醒的地方添加了一些译者说明。尽管我们在翻译过程中力图做得更好，但终究因业务水平、英文水平，乃至中文文学水平的欠缺，以及翻译过程中的粗心和不够严谨，致使本译作存在错误、不足和不当之处，也恳请读者批评指正，并热切期望读者对本书提出宝贵意见和建议。

译者
2014 年 12 月

数学科学表明的是原理，它是描述事物之间不可见关系的一种语言。

——Ada, Countess of Lovelace,
Quoted in Dorothy Stein (ed.):
Ada: *A Life and a Legacy* (1985)

科学的本质是：问一个不恰当的问题，于是走上了通往恰当答案的路。

——Jacob Bronowski:
The Ascent of Man (1973)

Hofstadter 定理：一件事情总是会花费比你预期更多的时间，就算是你已经考虑过本条 Hofstadter 定理。

——Douglas R. Hofstader:
Gödel, Escher, Bach: An Eternal Golden Braid (1979)

献辞

谨将本书献给我的妻子 Lindy 和女儿 Sarah、Jennie。我爱你们。

关于作者

C. J. Date 是一位独立作家、演说家、研究员、顾问，在关系数据库技术领域颇有建树。C. J. Date 以他的 *An Introduction to Database Systems*（第 8 版，Addison-Wesley，2004 年出版）一书而闻名，到写作本书时为止，该书发行量已达到近 90 万册，被全球数百所大学和学院使用。他还出版了许多数据库管理方面的著作，新近出版的包括下面这些著作。

- Ventus：*Go Faster! The TransRelations Approach to DBMS Implementation*（2002 年出版，2011 年再版）
- Addison-Welsy：*Databases, Types, and the Relational Model: The Third Manifesto*（第 3 版，与 Hugh Darwen 合作编写，2007 年出版）
- Trafford：*Logic and Databases: The Roots of Relational Theory*（2007 年出版）和 *Database Explorations: Essays on The Third Manifesto and Related Topics*（与 Hugh Darwen 合作编写，2010 年出版）
- Apress：*Date on Database: Writings 2000-2006*（2007 年出版）和 *The Relational Database Dictionary, Extended Edition*（2008 年出版）
- O’Reilly：*SQL and Relational Theory: How to Write Accurate SQL Code*（第 2 版，2012 年出版）、*Database Design and Relational Theory: Normal Forms and All That Jazz*（2012 年出版）和 *View Updating and Relational Theory: Solving the View Update Problem*（2013 年出版）

Date 先生于 2004 年成为计算机工业名人堂中的一员。他在用明晰易懂的方式解释复杂技术课题方面具有傲视群雄的能力。

前 言

关系数据模型是百年来最伟大的技术发明之一，它是我们完成数据库领域任何事情的基础。的确，它使数据库管理成为一门科学，而不再像过去那样是一些技巧、技术和经验法则的特定集合。因此，每一个与数据库管理有关的专业人员，或多或少都会主动去获得一些与关系模型有关的知识，以加深对关系模型的理解。因为如果没有它，想开展高效的工作、获得较高的工作性能几乎是不可能的。

不幸的是，想要达到如上所说的“获得知识，加深理解”是不容易的。这有多方面的原因，但影响最大的原因是SQL语言，它是一种“关系型”语言的官方标准[1]，当今市场上的每个数据库产品都支持其中的某些方面。大家普遍承认，至少是应该普遍承认，作为关系模型的抽象概念的某种具体实现，SQL 有着非常严重的缺陷（这是我在前面那句话中给“关系型”加上引号的原因）。至于在数据库世界，每个人之所以都知道一些关于SQL的知识，是几乎所有的关系型（或可能的关系型）教学的重点都倾向于SQL本身，而不是基本的理论。因此，仅仅因为他们知道 SQL，人们就认为他们知道关系理论，也就不足为奇了。然而，令人难过的是，如果你只是像这样知道 SQL，那么你肯定不知道关系理论，而你不知道的事情最终可能会伤害你。

本书的首要目的就是详细地教你关系理论（至少是尽我所能地详细），第二个目的是从关系理论的角度描述 SQL（或者描述 SQL 的核心特征），这也可能会给我带来一些其他灵感。一些读者可能知道，不久前我出版了另一本书来论述这件事情，书名为 *SQL and Relational Theory: How to Write Accurate SQL Code*（第 2 版，2012 年 O’Reilly 出版）。然而，与现在的这本书不同，早期的图书主要是面向数据库从业人员，通常情况下适用于至少有三四年的“工作在数据库系统第一线”的人员，或者经常把它作为日常工作一部分的人员（因此，肯定会有一些关于 SQL 的工作经验）。那本书的主要目的是展示如何把关系理论应用到使用 SQL 的过程中，因此，

1 多年来，SQL 已经演变了几个版本。在本书出版时的版本是 2011 版（即 SQL:2011）。规范的引用格式为：国际化标准组织（ISO），数据库语言 SQL，文档 ISO/TEC9075:2011。第 10 章和附录 E 对这种引用做了详细的介绍，第 10 章还专门简要介绍了它的相应发展史。

SQL 的行为看起来就像是一个真正的关系语言一样（在那本书中我引用的一个规则是，从关系型的角度使用 SQL）。在本书和那本书之间不可避免地会有一些重复。实际上，本书中有一些文字就是从早期写作的图书中复制和粘贴的，有些几乎是完全一样。不管怎样，你现在看到的一定是一本不同的图书，因为它是面向不同的读者（参看下面的详细解释）。然而，本书接下来的内容还需要频繁引用早期的图书，所有的引用都以简写的形式给出（即 *SQL and Relational Theory*）。而且，我认为如果你熟悉我的早期作品，那么你从本书中可能不会获得太多的知识。当然，我不是不鼓励你阅读此书，只是如果你读了，你可能不会发现太多的新知识。

适合的读者

本书面向的读者是计算机专业人员。前提是你了解一些通用的计算机和编程知识，也至少熟悉一种通用的编程语言。但是你不需要了解数据库，无论是关系型的还是其他的，也不用了解 SQL。当然，你至少要知道当今的数据库系统都几乎被假设为“关系型”（不管它是不是），但是我不会依赖于这个假设。然而，请注意，如果你恰好已经了解数据库的一些知识，那么你或许要格外留心本书中讲的内容！你可能会发现需要抛掉一些以前的想法（我们大家都知道，抛掉已有的想法是很难的）。当今的数据库产品即便被贴上了“关系型”的标签，但实际上却不完全是关系型的，它们在很多方面都与理论上的关系型思想相差很远，你很快就会发现这一点。正如我在 *SQL and Relational Theory* 一书中写到的那样：

> 在这里我要为我带有进攻性的口吻道歉。但是如果你对关系模型的了解仅仅来自于对 SQL 的了解，那么恐怕你并没有如你所期待的那样真正了解关系模型，你也许应该知道“有些事情不是这样的”。我也不能过于强调：SQL 和关系模型不是一回事。

注意：我想说的是，关于本书的详细材料可以参照我的一个在线论坛，如果想要更详细的资料，请登录网站，网址为：www.justsql.co.uk/chris_date/chris_date.htm，也可以参考视频，网址为：http://oreilly.com/go/date。

本书结构

本书分为三个部分及附录。第一部分为“基础知识”，主要介绍了关系模型本身，换句话说，介绍了关系数据库技术的理论基础（虽然它也强调了整个理论的实际应用）。第二部分为“事务和数据库设计”，主要讨论了为了理解通用的数据库和特定的关系型数据库而需要的知识，但是不需要真正了解关系模型本身（除非它以

该模型作为基础）。第三部分为“SQL”，又回到了第一部分的材料，展示了该部分讨论的概念是如何用 SQL 语言实现的。至于附录，是很多材料的混合（这也是附录通常的作用），我认为这不值得在书的正文部分过多描述，因此都放在了这里加以详细描述。

注意：复习题和练习是大部分章节中不可或缺的一部分，我建议你在查阅后面的答案之前尝试着回答这些问题，以使自己得到训练。尤其你要知道的是，经常有目的地做些练习，可以在很多方面得到锻炼。

致谢

我想再一次感谢我的妻子 Lindy，她在本书的整个著作过程中给予了大力支持，同时要感谢所有的前辈。我还要感谢 David Livingstone，是他最初给了我写本书的一些建议或类似的想法，也要感谢 Allen Noren，是他促使我写作本书的（在我的其他的由 O’Reilly 出版的著作中，Allen 的想法为此奠定了坚实的基础，就像本书一样，也是奠定了理论基础。我相信他是对的，这就是我想要写的著作）。最后，我还要感谢在线论坛的讨论者，因为人物众多，不胜枚举，他们的问题和评论帮助我合理地组织了本书的结构，也帮助我决定了本书内容的取舍。当然，无论是事实还是判断中的任何错误，我都会一如既往地负责。

C. J. Date
加利福尼亚希尔兹堡
2013 年

目　录

第一部分　基础知识

第一部分

基础知识

第 1 章

数据库基本概念

我们的生活被琐事浪费掉了……简化，简化。

——Henry David Thoreau: *Walden*（1854）

本章是一个介绍性的概述，目的是提供一个距离我们非常遥远的观点。它故意没有讲解得很深奥，如果你已经了解了关于数据库管理的一些知识，也许会发现本章内容都已熟悉。但是我想你应该花些时间把本章从头到尾通读一遍，这是非常值得的，如果只是想获得背景知识，可以看后续的章节。同时，本章还介绍了一些可以运行的示例，在后面的章节中我们也一定会遇到这些例子，并逐步熟悉。

1.1 什么是数据库

数据库被认为是一种电子文件柜，它包含了一些数字化的信息（即数据），这些信息可以被永久保存在某种存储介质上，通常是被保存在磁盘上。用户可以通过管理数据库的软件发出请求（request）或命令（command）来向数据库中插入信息，删除、修改或检索数据库中已经存在的信息，这种管理数据库的软件叫做数据库管理系统（DBMS）。

注意：

在本书中，术语user的含义将根据上下文的要求理解为应用程序员或者是交互用户[1]或者是应用程序员和交互用户。

实际上，现在这些发给 DBMS 的用户请求可以采用各种各样的方法进行格式

1 这里仍然指的是计算机专业人员，而不是一个天才的“终端用户”，他可能会合理地忽略掉本书中讨论的大部分内容。

化（例如，单击鼠标）。然而，为了达到我们的目的，采用某种正规语言中的简单文本串的形式来表示这些请求会更方便些。例如，有一个人力资源数据库，我们可能会这样写：

```
EMP WHERE JOB = 'Programmer'
```

这个表达式表示了一个检索请求（retrieval request）——但通常我们都称之为查询（query）——这个表达式的含义是要获得员工工作性质为 Programmer 的员工信息。

可以运行的例子

图 1.1 给出了某种类型数据库中的一些样本数据，包含供应商表（suppliers）、零件表（parts）、货物表（shipments，即由供应商提供的零件数量）。

S

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

PNO	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

图 1.1　供应商表和零件表数据库——样本数据

从图 1.1 可以看出，这个数据库包含了三个**文件**（file）或称之为**表**（table）。（实际上它们就是**关系**’［relations］，我们将在第 2 章讲解，但在本章中我们称之为“文件”或者“表”。）这些表的名称分别为 S、P 和 SP。因为它们都是表，所以它们都是由行和列组成的（在传统的文件术语中，行对应为文件中的记录，列对应为字段）。可以按照如下的方式来理解。

- 表 S 代表**签署了合同的供应商**（suppliers under contract）。每个供应商都有一个唯一的供应商号（SNO）、姓名（SNAME）（姓名必须唯一，图 1.1 中给出的样本数据中姓名唯一只是巧合）、一个状态值（STATUS）、供应商位置（CITY）。注意：在本书的其他位置，“签署了合同的供应商”都缩写为**供应商**（suppliers）。
- 表 P 代表**零件的种类**（kinds of parts）。每种零件都有一个唯一的零件号码

（PNO）、零件名称（PNAME）、零件颜色（COLOR）、零件重量（WEIGHT）、零件储存位置（CITY）。注意：在本书的其他位置，“零件的种类”都缩写为零件（parts）。

- 表 SP 代表供应关系（shipments），即表明零件从哪里运输或者由哪个供应商来提供。每个供应关系都有一个供应商号（SNO）、一个零件号（PNO）以及供应数量（QTY）。因为在特定的时间内，一个特定的供应商和特定零件之间至多有一种供应关系，所以由供应商号和零件号组合在一起表示一个唯一的供应关系。

本书中其他地方的例子大多是基于前面提到的这个数据库。现在，你可以再仔细看看前面的这个数据库，我已经在很多其他的书籍和著作中用过，包括*SQL and Relational Theory*[1]，在许多现场演讲时也用过，所以你可能会有点厌烦的感觉。但是就像我在别处写的那样，我相信在各种不同的出版物中使用相同的一个例子对学习是有帮助的，而不是障碍。当然，真正的数据库应该比这个像“玩具”一样的例子复杂得多，但是使用接近于实际的例子会遇到的麻烦就是它们太复杂了，在某种程度上会造成只见树木不见森林的结果。所以我还是要说，即使这个例子有点不现实，但供应商和零件表数据库至少适合说明我们后面要测试的所有观点。因而，要达到本书的目的，这个例子足够了。但要注意的是，在本书其余部分的例子中，除非做特殊说明，否则我会假定使用图 1.1 所示的特定的样本数据。

1.2　什么是数据库管理系统

从现在开始，每当我提到“给出一个典型的数据库”，如图 1.1 所示，意味着给出一个如用户想象的那样的数据库（有时称之为**逻辑数据库**［logical database］）。逻辑数据库与物理数据库是相对而言的，物理数据库是被**数据库管理系统**（DBMS，Database Management System）所理解的（即它是实际存储在计算机中的数据）。因此，需要进一步强调的是，这里所谓的逻辑和物理数据库并不是两个完全不同的事情，相反，它们是从不同视角来看待的同一事情，如图 1.2 所示。

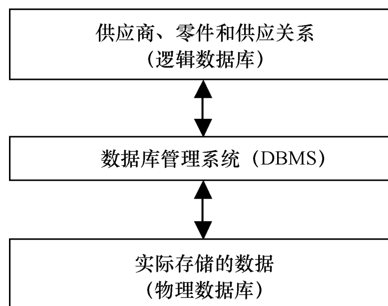


图 1.2　数据库系统结构

1　本书的前言中曾提醒读者，在本书中我将用缩写形式“*SQL and Relational Theory*”代表我的著作 *SQL and Relational Theory: How to Write Accurate SQL Code*（第二版，O’Reilly, 2012）。

从图 1.2 可以看出，DBMS（即管理数据库的软件）作为系统逻辑层和物理层之间的一个媒介，可以有效地提供以下服务：用户向数据库发出请求的格式可以依据逻辑数据库的形式来标识，然后依据对应的物理数据库，DBMS 会一一实现这些请求。

DBMS提供的一个通用功能就是向用户隐藏系统物理级别的实现细节（大多数的程序设计语言也是在物理级别向用户隐藏实现细节），换句话说，DBMS给用户提供了一个更抽象的数据库概念，相比物理数据库级别而言，这种方式会更友好。例如：以实际存储在系统中的方式来表示数据库概念[1]。

现在我们知道了 DBMS 是一个复杂的软件，它包含很多构件。但是有一个构件我现在必须提一下，因为它是这本书的主题，它就是优化器（optimizer）。优化器是 DBMS 的构件之一，其主要职责是决定如何正确执行用户的请求。大多数请求（实际上几乎是所有的请求）都能够通过各种不同的方法来实现。而且，这些不同的方法一般在性能特点上有很大区别。特别是在执行时间上它们确实有很大区别，可以从几微秒到几天。因而，对于优化器来说，选择一个“好”方法去实现特定的请求是非常重要的，这里的“好”实际上是指好的性能（good performance）。

前文中存在一个直接而且重要的暗示就是假设优化器能很好地发挥它的作用，用户根本不必考虑性能的问题。但有一个事实我必须在这里声明一下（这是突然在我脑海中闪现的一个想法），记录一直是关系模型中的一个主要目标，性能问题应该是系统所担忧的，而不是用户应该关心的。从某种程度上来说，如果这个目标没有达到，就说明系统是失败的（或者确切地说，至少是不成功的）。

1.2.1　数据依赖

逻辑数据库和物理数据库是有区别的，要严格进行区分，但这个事实也允许我们实现一个重要的目标，就是数据独立性（data independence）。数据独立性（顺便说一下，这其实不是一个很恰当的数据，但似乎我们也只能坚持使用它）意味着我们可以随意改变数据在系统中的存储和访问方式，而对于用户看到的数据库却不需要做任何相应的变化。其实我们可能想改变数据库存储或访问方式的原因就是性能（performance）。我们做这样的改变而不用去修改用户看到的数据库，这意味着已存

---

1 顺便说一下，虽然物理数据库不如逻辑数据库抽象，但它仍然是经过抽象后形成的。它一般包含很多成分，如存储的文件或索引，这些存储的文件或索引以一种更低级别的方式来表示，如页或磁盘空间。页或磁盘空间相应地又是一些更低级别组成成分的抽象，如二进制位或字节，当然，它们本身也是一种抽象（抽象的级别可以很深，只要你能想像的到）。

在的一些应用程序、查询等都可以在变化之后仍然按照原来的方式工作。因此，非常重要的一点是数据独立性可以保护已存在的资产，这些资产可以存在于已有的应用程序、用户培训、数据库设计或者其他事情中。

1.2.2 DBMS 的其他功能

再重复一下，DBMS是逻辑数据库和物理数据库之间的一个媒介。换句话说，它支持通向数据库的接口（interface）（在本书中的其他地方，我一直不加限制地使用数据库［database］来专门表示逻辑数据库，除非在上下文环境中另有要求）。因此，DBMS 的主要职责包括：（a）负责接收用户请求，这些请求可以是查询或者修改，具体形式要根据逻辑数据库的要求来决定；（b）通过解释或执行的方式对这些请求做出响应，换句话说，就是根据物理数据库的形式来执行这些请求。注意：在极少的情况下 [1]，术语修改（update）一般用来指的是插入新数据、删除或修改已存在的数据等这类请求。

因此，DBMS 可以“保护用户已有的数据”（例如，可以保护数据在系统内部真正的存储形式等细节）。我们可以有些得意地说，它保护了用户数据。但是我的真正意思是，它提供了安全性（security）、并发 性（concurrency）、完整 性（integrity）和恢复等（recovery）控制。简要说明如下。

- 安全控制：用来保证用户请求的合法性。即有疑问的用户正在请求一个操作，允许他或她来操作有权利访问的数据。例如，在供应商和零件数据库中，某些用户可能不会被允许查看供应商状态数据；某些用户可能根本就不允许查看供应商表；某些用户可以被允许查看伦敦的供应商，但不允许查看其他城市的供应商；某些用户可以被允许检索供应商信息但不允许修改，等等。简而言之，必须限制用户执行那些他们允许被执行的操作。注意：安全当然是很重要的，但对安全控制的更多细节介绍已经超出了本书的范围（在第 7 章中将给出一个简短的描述）。
- 并发控制：就是要处理同一时间可能会有多个用户使用同一数据库的问题。假设你要查询数据库中是否有供应商 S1 供应的商品，得到肯定回答后继续提问供应的零件数量平均值是多少，当被告知根本没有这样的供应关系时，这是一个让人很讨厌的事情。推测一下，可能是因为一些其他用户删除了这些数据。并发控制的目的就是要处理这样的问题，我将在本书的第二部分对其进行详细讲解。
- 完整性控制：就是要保证数据库中的数据是正确的（因为在某种程度上提

---

1 这种极少的情况是很重要的，参见本章后面的练习 1.2。

供这样的保证是可能的）。例如，要向供应关系的表中插入一条供应商 S6 的供应关系记录，如果供应商表中没有 S6 供应商，则这个请求一定会被拒绝。同样，要把供应商 S1 的状态值修改为 200 时，如果这个状态值规定不能超过 100，则这个请求也会被拒绝。这些例子还不足以说明完整性控制是相当重要的，我将在第 3 章和第 6 章中对其进行详细讲解（在第 13 章也会涉及完整性控制）。

- 恢复控制：就是假设数据库从不会忘记任何事情，它会一直记得所有的事情。也就是说，一旦数据被插入到数据库中，就再也不会被删除了（除非用户有特别明确的要求），即使发生一些失败的操作，如系统被破坏或者磁盘被破坏。我将在本书的第二部分对其进行讲解。

最后，关于DBMS，我还要说一件事情。从我前面讲解的每一件事情中可以看出，有一点是相当清楚的，那就是数据库之间的逻辑结构是有区别的，它是存储数据的仓库，而DBMS是管理这个仓库的软件。不幸的是，在数据库领域使用术语数据库（database）表示DBMS已经很普遍了[1]，但我想强调的是，这个术语已经非常非常普通了，在本书中我不会再采用此种说法。因为如果要把DBMS称为数据库，那真正的数据库又是什么呢，这是个问题。

1.3 什么是关系型 DBMS

现在我们已经知道了什么是 DBMS，但什么是关系型 DBMS 呢？当然，首先它是一个 DBMS，它要提供本章介绍的 DBMS 具备的所有功能：数据存储、查询、修改、恢复控制、并发控制、安全控制和完整性控制，以及本书中没有讨论的其他功能。但是第二点要注意的是，它又是关系型的，这就意味着用户接口在真正实现时必须是关系模型。换句话说，关系模型被看作是 DBMS 中实现用户接口的一种方法。

在继续讲解之前，还要强调一点，这个方法（即关系模型）其实很简单！关系模型的这些方法并不是紧身衣，相反，它们是一种规则，对用户来说可以是生活更加容易的一种规则（在某些方面，对系统来说也更简单，但是重点是针对用户的）。现在，虽然这种方法有时看上去稍复杂些，但最重要的是，关系模型是很正规的。既然正规就需要有正规的术语，正规的术语有时会令人却步。但是术语中的这些定

---

1 一个典型的例子，大家几乎可以在 Internet 上随处看到这样一种现象："MySQL 是世界上最受欢迎的开源数据库"。但是它不是，它也许是世界上最受欢迎的开源数据库管理系统（我也不太清楚），但不是数据库。

义实际上相当简单（别忘了，供应商和零件数据库就相当容易理解，不是吗？）。事实上，我们谈论的这些定义要比过去使用的类似定义简单得多，就像IMS和IDMS这样的前期关系系统或非关系系统[1]。

再重复一下，关系型数据库管理系统是支持真正实现关系模型的用户接口的DBMS。对用户而言，其含义如下。

- 数据看上去是关系型的。
- 可以使用关系运算符（如，采用关系运算来处理数据）作为检索请求和修改请求的基础。举个简单的例子，如：

```
S WHERE CITY = 'London'
```

这个表达式表示查询供货地点在 London 的供应商，它使用了关系型的“限定”运算符。规范的表达就是，它请求在表 S 中查询地点在 London 的供应商集合。

然后，在本书的第一部分中，我们将仔细看看“数据看上去是关系型的”的真正含义，我们也要检验各种关系运算符，看它们是否能起到作用。然后，要注意的是，这种解释也不一定是非常详尽的（这些主题的详尽解释可以参照 *SQL and Relational Theory*），但是就现状来说，它是非常综合的，也是非常准确的。

不幸的是，这里还有一个问题。为了说明这些定义，我们还要讨论一些问题，即我需要明确给出一些编码实例。为了表达这些实例，我需要采用正式的语言来描述，但是关系模型没有规定任何一种这样的语言，而且它是在很高的抽象级别上定义的，原则上是能够来具体实现任何不同的语法形式。现在，一种标准的、具体的语言确实存在了，即 SQL，目前在市场上它也或多或少地获得了所有主流数据库产品的支持。然而就像在前言中提到的，SQL 是有很多缺点的，比如它的复杂性、不完整性、难掌握，在很多方面还容易造成误导。因此我计划编写本书的目的如下。

- 第一，我会在根本不使用SQL的情况下来解释关系模型（见本书的第一部分）。作为替代，我使用了一种叫做**Tutorial D**的语言，它是专门为此目的设计的一种语言。注意：我相信**Tutorial D**是一种可以自我说明的语言。然而，如果需要更容易理解这种表述的话，可以参考*Databases, Types, and the Relational Model: The Third Manifesto*（第三版，Addison- Wesley,

1 这要比各种建议中提到的类似定义简单多了（但这种犯错太频繁了，这样说我感到非常抱歉），这是因为有关系模型作为替代（例如：XML、NoSQL、角色模型等）。顺便说一下，我从来没有看见过这样一种建议，即提出这种建议的人真正理解了关系模型。可以肯定的是，如果你想声明技术 A 不好，需要用技术 B 替代，那么对于你来说首先理解技术 A 就是一项义不容辞的职责，尤其是要证明技术 B 如何解决了特定的问题，而技术 A 不能解决。

2007）[1]。

- 第二，我将展示关系模型的思想是如何具体用 SQL 实现的（见本书的第三部分），因此可以肯定的是，我并不是要在本书中全面介绍 SQL 语言，只要够用即可，即我在前言中提到的语言的核心特征。

（第二部分是对前言中提到的问题所做的简短说明）注意：也许我应该声明一下，由于前面提到的计划，接下来进行的工作可能与目前市场上的大部分书籍或报告不太一样，所以我打算叫做“关系数据库的介绍”。

在结束本节前，再说明一点。你也许知道，但也许不知道（但是我希望你知道）：关系模型最早是由 E.F.Codd 发明的，当时他还是 IBM 的一名研究人员（E 代表 Edgar，F 代表 Frank，但他一直喜欢用首字母签名，作为他的朋友，我也感到很骄傲）。1968 年末，Codd 已经成为了一名数学家，他第一次实现了数学定律可以用来把一些固有的原理用在某一领域中，例如，数据库管理，而在以前这些都是做不到的。他在关系模型上最初产生的一些思想可以在 1969 年 IBM 的研究报告中找到（进一步的研究可以参见附录 E）。

1.4 数据库系统与程序系统

这两个概念好像不太好区别，但是也有一些相似性。事实上，通常不认为一个数据库系统就是一个程序系统（更确切地说，是一种特例）。图 1.3 给出了代码片段，这段代码的目的是要计算一维数组 A 中整型数据的和，并将其显示在终端上（该片段采用一种假设的语言来表示，但该语言具有自解析性）。

```
VAR I , N, SUM INTEGER ;
VAR A ARRAY [1..N] OF INTEGER ;

SUM := 0 ;
I := 0 ;
DO WHILE I < N ;
     I := I + 1 ;
     SUM := SUM + A[I] ;
END DO ;
DISPLAY 'The sum is ' || SUM ;
```

图 1.3 代码片段

---

1 在本书第一版出版之后，Tutorial D 已经修订并扩充了一些内容。修订版本的描述（在本书中我采用的就是修订版）可以在 Hugh Darwen 和我的另一本书中找到，即 *Database Explorations: Essays on The Third Manifesto and Related Topics (Trafford, 2010)*，也可以参见网页：**www.thethirdmanifesto.com**。

相关说明如下。

- **语句**：整个代码由 9 条语句组成。在程序设计语言中，一条语句（statement）就是一个指令，它可以引发一些动作，比如，定义或修改一个变量、改变控制流等。通过观察你会发现语句与表达式之间的逻辑差异，表达式就是由一个值构成的（可以认为它是计算或决定值的一种规则）。例如，在图 1.3 中，“I=I+1”是一条语句（即赋值语句），而 I+1 是一个表达式。注意，在整本书中，我都采用通用的语法来表示，即语句以分号结束，而表达式不用。
- **类型**：类型（type）就是一组值的集合，即特定类型的所有合法值的集合。任何一个值或者变量都要属于某种类型 [1]。图 1.3 所示的代码片段涉及 3 种类型：**INTEGER**（即所有整数的集合）、**ARRAY[1..N] OF INTEGER**（即下限为 1、上限为N的一维整型数组）、**CHAR**（所有字符的集合）。注意，就像我们一会儿将要看到的（看下面将要讨论的比较运算符），它也涉及了**BOOLEAN**，即所有逻辑值的集合。当然，它只有 2 种取值，即**TRUE**和**FALSE**。
- **变量**：变量（variable）就是所有值的容器（通常不同的时候会有不同的值）。图 1.3 所示的代码片段涉及了 4 个变量，即 **I**、**N**、**SUM**、**A**。给定的变量的当前值（指的是在特定的时间，这个变量所包含的值）是可以用一种而且只能用一种方法来改变，即执行一条赋值语句，在该语句中把变量作为赋值对象。事实上，变量都是可以被赋值的，而值也都可以被赋予变量。
- **赋值**：赋值（assignment）（比如图 1.3 中的:=）就是一个运算符，它可以修改一个变量的值。也就是说，给变量赋予一个值，这个值也许和以前的值不相同。图 1.3 所示的代码片段包含了 4 条赋值语句。
- **符号**：符号（literal）是一种“自定义符号”。例如，符号可以表示某个值，而这个值的类型就由讨论的符号来决定。图 1.3 所示的代码片段中有 3 种符号，即 0、1 和‘**The sum is**’（前两个都是整型，第三个是字符型）。
- **值**：值（value）指的是单独的一个常量。它通常由表达式来声明，也可以被赋予一个变量。注意在特定情况下，符号和变量引用都是表达式，因此，它们都可以用来声明值。每一个值也表达一种特定的数据类型。
- **只读运算符**：只读运算符（read-only operator）就是一种运算符，比如“+”，它可以修改旧值，得到新值。例如表达式 2+3，就从原来的值 2 和 3 得到了一个新值 5。但要特别注意的是，当它被借用时，只是返回一个结果而

1　事实上，是属于一种特定数据类型，除非有类型继承关系，但在本书中是不存在类型继承关系的。注意：类型是不能相交的（至少我们所关心的就是），即没有一个值会同时属于 2 个或者多个类型。

不修改任何事物（尤其是不修改它的操作数[1]）。但还要引起注意的是，我在早前所说的那些就是一种计算值的规则，但表达式可以等价地去表示一种只读运算符的借用。事实上，术语表达式（expression）和只读运算符（read-only operator）借用是可以互换的。最后要注意的是，只读运算符不能使用像“+”这样的专门运算符来声明。事实上，大多数这样的运算符都采用了一些惯常使用的标识符来声明。例如，许多程序设计语言都支持**RANDOM**这个只读运算符来产生一些随机数。

- **比较运算符**：比较运算符（comparison operator）就是类似于“<”的运算符，当被借用时，返回一个真值（**TRUE** 或者 **FALSE**）。事实上，这样的运算符也是只读运算符，但它们的返回值类型为 **BOOLEAN**。

最后要说明的是，之所以详述这些大家已经相当熟悉的知识，是因为所有以前出现的概念都是和数据库直接相关的，这些在接下来的内容中将会看到。

更多的类型

首先，对于类型的定义，我尤其要多讲一些。图 1.3 所示的代码片段不能解释这一点，但是通常情况下，类型或者由系统定义，或者由用户定义，它们也可以任意组合。例如，在一些几何应用中，用户也许会定义如下类型：**POINT**、**LINE**、**RECTANGLE**、**CIRCLE** 等。然而，对于只是使用它们的用户（与实际定义它们的用户是相对而言的）来说，这些定义类型的用户更能准确地去认识系统。所以，在这本书后面我所提供的例子中，为了尽量减少失去或不失去通用性，我限定大部分情况下只能定义的类型有 **INTEGER** 和 **CHAR**，但不完全是这样。

其次，我曾经说过，每个值都是属于某种类型的。它要把每个变量、每个参数赋予每个运算符、每个只读运算符或者每个表达式（尤其是每个符号和每个变量引用），这些运算符也是属于某种类型的，因为有这些结构存在，所以使用它们时肯定会声明某些值。具体说明如下。

- 就变量、参数和只读运算符来说，当定义问题中的一些结构时，问题中的类型就要被说明。例如，看图 1.3 中的变量定义，即 **VAR** 语句。
- 就表达式而言，当要计算问题中的表达式时，它的类型只是返回结果的类型。例如，表达式 2+3 的类型为 **INTEGER**，这是因为该表达式的结果 5 是 **INTEGER**。

---

1 这样就会出现一个很明显的问题，即是否还有类似于这样特性的运算符，但它是不是只读的呢？答案当然是肯定的。修改运算符就是在被借用时不返回值，但是修改一些变量。然而，我们将会在后面看到，任何给定的修改运算符的借用在功能上都等价于一个赋值操作。从逻辑上讲，赋值就是我们需要的修改运算符。

第三点，也是最后一点，理解与给定类型 T 的关系，这一点很重要。对于类型 T，定义了一组操作符的集合，要操作类型 T 中的值和变量（因为没有操作符的类型是无用的）。例如，对于类型 **INTEGER** 而言，为简单起见，我只定义系统，由代理负责定义类型。换句话说，该系统必须定义如下内容。

- 为了给整型赋值或者比较整型，必须定义运算符“:=”、“=”、“<”等。
- 为了在整型上执行算术运算，必须定义运算符“+”、“*”等。
- 为了把整型转化为字符串，也要定义 **CAST** 运算符。
- 不要定义“||（连接运算符）”、“SUBSTR（求子串）”这样的运算符（至少在我给的例子中没有使用），因为这些运算符对于整型数据操作没有意义。

理解给定类型 T 中必须包含赋值和等式子、比较是非常重要的，而且，这些运算符的语法必须要满足下述条件。

- 赋值：把值 v 赋值给变量 V 之后，比较 *V*=*v* 就是 **TRUE**。注意，这个条件有时也称为赋值原理。
- 等式：如果 *v1* 和 *v*2 具有相同的值，则 *v*1=*v*2 就是 **TRUE**（提示：它们必须要具有相同的类型）。注意下面这条重要推论：如果存在运算符 *Op*，且 *Op*（*v*1）≠*Op*（*v2*），那么 *v*1=*v*2 就是 **FALSE**。

注意：对于“:=”和“=”这两个运算符，它们的等价性是非常重要的。因为没有它，我们就不能讨论值 *v* 和值的集合 *S*，不管 *v* 是否出现在 *S* 中（例如，判断 *v* 是否是 *S* 的元素）。

最后，我再重复一下，之所以详细讨论我们已经相当熟悉的知识，是因为所有的定义都是直接与数据库相关的，这一点在下面的章节中将会看到。

1.5　练习

现在到了该做练习的时候了。当然，在本书中的第 1 章就进行查找练习是不可能的，下面大多数是复习题。尽管如此，我还是建议你们尽力独自去回答问题，而不要去看后面给出的答案。注意，前两个练习似乎有些不公平，因为我还没有提供足够的知识让你们去正确回答它们，但是我认为你们应该了解一下，它们并不难。

1.1　下面的 **Tutorial D** 表达式的含义是什么？

a. `P WHERE WEIGHT < 12.5`

b. `P { PNO , COLOR , CITY }`

1.2　写出 **Tutorial D** 下面语句的功能。

a. `DELETE S WHERE STATUS = 10 ;`

b. `UPDATE S WHERE STATUS > 10 : { STATUS := STATUS + 5 } ;`

1.3 什么是数据库？什么是 **DBMS**？

1.4 解释下列术语：（a）安全控制；（b）完整性控制；（c）并发控制；（d）恢复控制。

1.5 什么是 **SQL**？什么是 **Tutorial D**？

1.6 用自己的话解释下列概念。

- 类型
- 值
- 变量
- 符号
- 赋值
- 比较
- 只读运算符
- 修改运算符

1.7 （不参考本章内容的前提下独立完成）供应商和零件表数据库中包含哪些表？都有哪些列？注意：该练习的重点是让你尽可能熟悉数据库的结构，至少是采用通用的术语来理解，但不需要去记住数据库中存储的数值。

1.6 答案

1.1 a．找出重量小于 12.5 的零件。这个例子中用到了关系限定运算符。b．给出每个零件的编号、颜色和所在城市。这个练习确实有些不公平，但也许你能猜出答案。正如限定运算可以选出特定的列一样，该例子中给出的投影操作也可以选择出特定的列。我们将在第 4 章给出进一步的解释。

1.2 a．删除所有状态等于 10 的供应商。b．把状态大于 10 的供应商的状态值都增加 5。顺便说一下，注意该例子中关键词 **UPDATE** 的用法。也许你有一点儿糊涂了，在数据库领域中，使用大写形式如 **UPDATE** 来指代改值操作的运算符已经成为一个标准（与之对应，**DELETE** 代表删除已存在的数据，**INSERT** 代表插入新数据），小写形式 update 代表运算符集合 **INSERT**、**DELETE**、**UPDATE**。因此，在本书中，当用运算符 **UPDATE** 时，我将采用大写形式。对于 **INSERT**、**DELETE**，则不会产生歧义，如果使用大写形式，有时会觉得多余，特别是在它们只是作为一个修饰语存在的时候，例如“**INSERT** 语句”（是否可以使用“insert 语句”）。因此，我决定在本书中这两种形式都采用，但要根据上下文环境来区分其具体的含义。

1.3 数据库就是采用电子化的形式存储数据的仓库。DBMS 是一个软件系统，它管理数据库并可以访问这些数据库。

1.4 安全控制就是保护数据库不受到非法操作；完整性控制就是保护数据库接受经过认证的、有效的操作；并发控制是保护用户的操作，使其不产生相互影响；恢复控制是保证数据不丢失。

1.5 SQL 是一种可以和关系数据库进行交互操作的标准化语言。目前，市场上的主流数据库都支持此种语言。Tutorial D 是一种专门为解释关系型概念而设计的语言，原型实现确实是存在的（可以参见网站 *www.thethirdmanifesto.com*），但是，在写本书的时候，还没有形成商业性的产品。

1.6 类型就是一组值的集合。一个值就是一个独立的常量，例如，整数 3 是不能改变的。变量就是一些值的容器，它可以被改变。例如，它的当前值是可以被另一个值所替代的。一个符号就是一个自我定义的一种标志，用来表示一个值。例如，数值 3。赋值是一种运算符，它可以改变一个变量的值。比较就是一种只读运算符，它可以在两个值之间进行对比，然后返回一个布尔值。更新运算符可以修改某些变量的值，换句话说，它也是一种赋值。注意，在练习 1.2 的答案中提到的运算符 INSERT、DELETE、UPDATE 都是关系型的更新运算符，因而都采用关系型的赋值来定义。进一步的解释说明可以参见第 2 章。

1.7 参考图 1.1。

第 2 章

关系和关系变量

没有什么可以像关系一样去处理问题。

——写给 William Makepeace Thackeray 的歉语

Vanity Fair（1847~1848）

在这一章中，我想向大家清楚地解释在第 1 章中提到的数据“看上去是关系型的”的真正含义。换句话说，我想要确切地解释什么是关系。首先，我需要向您展示一些曾在前面章节讲过的事情，即（a）关系模型是最精确的；（b）这种精确需要精确的或规范化的术语；（c）这种规范化的术语让人感觉有点畏惧（它真的会成为理解过程中的一个障碍）。但是这些术语中提到的概念却是相当直观的。在你和这些规范化斗争的时候，我希望你已经做好准备去付出必要的努力。

2.1 关系

我们再重温一下供应商和零件数据库（如图 2.1 所示，它和图 1.1 所示是一样的），从关系型的视角来看，以前在数据库中被称作的“文件”或“表”其实都是关系。现在，关系就是一个数学上的概念，它有一个非常精确的数学定义，在本章结束时我将会给出这个定义，现在我们知道关系被描述成表的形式就足够了。每一种关系都由标题和内容组成，即：

- 标题是由一些属性（attributes）构成（在表格中用列表示）；
- 内容是由一些元组（tuples）构成（在表格中用行表示）。注意：元组有时也被称作偶对。

S

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

PNO	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

图 2.1 供应商表和零件表数据库——样本数据

下面我们就仔细看看什么是属性和元组。

2.1.1 属性

属性由名字和相应的类型组成（更规范地说，属性是由属性名和类型名组成的偶对）。例如，供应商关系有一个属性，它的名字为 **STATUS**，它的类型名为 **INTEGER**，这就意味着属性的合法值都是 **INTEGER**，而且只能是 **INTEGER**。

在这个定义中我要强调下面两点。

- 这里我们可以把属性的类型（type）也称为域（domain），即类型和域指的是同一件事情（在早期写的关系型的著作中都称之为域，最近都用类型替代了，所以在本书中我坚持使用类型）。
- 图 2.1 中给出的关系的表格形式并没有给出属性的类型。例如，属性 **STATUS** 的类型为 **INTEGER**，但在图中并没有指示出来。但是不要忘记，从概念上来说它是存在的。

供应商关系中的属性如下所示：

```
SNO     : SNO
SNAME   : NAME
STATUS  : INTEGER
CITY    : CHAR
```

上面给出的每一对中的第一个名字是属性名，第二个是相应的类型名，在此例中我假设类型 SNO（指的是供应商编号）和 SNAME（指的是供应商的名字）是用

户已经定义的类型，类型 **INTEGER**（所有合法的整型值的集合）和 **CHAR**（所有合法的字符串的集合）是系统定义的。但是，现在我要提醒您的是，在第 1 章中提到这个例子时，用户定义的类型是假设被看作是系统已经定义好的（至少对于使用它们的用户来说是这样，但对于真正定义它们的用户来说不是这样的）。所以，从这一点上来说，我还要进一步简化这个例子，假设属性 **SNO** 和 **SNAME** 都是 **CHAR**（这个类型是系统已经定义的）类型的，即：

```
SNO     : CHAR
SNAME   : CHAR
STATUS  : INTEGER
CITY    : CHAR
```

事实上，在本书中对于用户定义的类型我没有解释太多。这种简化可以允许我忽略很多细节，而这些细节对于定义类型的人来说是很重要的，但对于使用类型的人是不重要的（对于关系型数据库来说也不重要）。

因此，供应商关系的标题就是一组集合，即：

```
{ SNO CHAR，SNAME CHAR，STATUS INTEGER，CITY CHAR }
```

这就是 **Tutorial D** 的标记方法。就像你看到的那样，在 **Tutorial D** 的标题中，我们使用一个或者多个空格代替冒号来分隔属性名和类型名；使用逗号（后面可以带有 0 个或多个空格）来分隔每个属性；使用大括号（“{ ” 和 “ }”）来把这些属性对封装起来。在论文中对于集合的常用表示就是采用大括号来封装元素，而元素是采用逗号分隔的。

语法说明：前面的这个段落包含了本书中第一次提到的有用的术语(commalist)，而且后面还要多次使用。它的定义如下：假设 ***xyz*** 表示一种语法结构 (例如：表示属性)，那么术语 ***xyz commalist*** 就代表 0 个或多个 ***xyz*** 的序列，该序列中的每一对相邻 ***xyz*** 的都用逗号分隔，在逗号之前或之后可以添加一个或多个空格。

给定标题中属性的个数称为度（degree），因此，我们可以把供应商和零件数据库中供应商关系的度称作为 4、零件关系的度为 5、供应关系的度为 3。注意：如果关系 r 的度为 n，那我们就称它为 n 元的。如果 n=1，则称为一元关系；如果 n=2，就称为二元关系；如果 n=3，就称为三元关系；以此类推。

最后一点要说明的是，就像我们前面看到的那样，一个属性一经定义，就必须是属性名和类型名的偶对。然而，在非正式的场合中，我们通常都是以属性名来单独标识属性，比如说 **STATUS** 属性（位于供应商关系中）。同样，我们在表示标题时，通常用属性名的集合来代替属性名和类型名的偶对，因此供应商关系就可以表示为{SNO，SNAME，STATUS，CITY}。这种简化或者简写可以节省很多多余的话。而且，这也是符合语法的，因为还有另一条规则（在这之前没有提到过，但是

通常都是这样理解的），即在某种程度上，在给定的标题中，不允许两个不同的属性使用相同的属性名。

2.1.2 元组

给定关系中的每一个元组都要对应于该关系中的标题，这样每个元组就会包含一个可用类型数值，对于每个属性都是如此。例如，供应商关系的主体就是元组集合，即供应商 S1、S2、S3、S4 和 S5，这个五元组中的每个元素都包含一个值，这些值对应于属性 SNO、SNAME、STATUS 和 CITY。而且这些值都是来自于可以使用的类型中的值，分别为 CHAR、CHAR、INTEGER 和 CHAR。注意：其实还可以采用更简单、更准确的一种方式表示，即把这些元组中的每个成员都赋予一个相关的标题，这是因为元组实际上就是有标题的，就像关系中定义的那样。

给定关系中元组的数量叫做关系的基数（cardinality），该定义既适合于给定的数据库，也适合于该数据库中的每一个关系 [1]。因而，我们可以说图 2.1 给出的供应商和零件数据库中供应商关系的基数为 5、零件关系的基数为 6、供应关系的基数为 12。注意：如果基数为 0，则该关系就是空的。一个空的关系就像是不包含任何记录的文件一样。

因此，我说偶尔也会发生关系中不含有任何元组的现象。它包含一个定义体，反过来定义体中也包含元组。但实际上我们通常讨论的关系中都含有元组。

2.1.3 关系的特点

关系和元组我们就讨论到这里，现在准备讨论关系本身。我想要说的最重要的一点是，关系是非常精确的、被规范定义的，因而，它们享有大量的规范的特点，无论从理论上还是从实际上来说，这些特点都是相当重要的。注意：就像我在别的书中写到的那样（例如：*SQL and Relational Theory* 一书），它让我相信理论是可行的！从其自身来看，关系理论的目的不仅仅只是作为一种理论，而且，该理论允许我们建立 100%可行的系统。这就是我非常相信它的原因，特别是在关系型的环境中，离开这些基础理论就是一个天大的错误。所谓的“关系型”系统至少是有一点吸引力的，但实际上，从这一点上来看，他们已经偏离了基础理论。

尽管如此，还是要讨论关系的 4 个特点，我先列出它们，然后再一一进行解释。

- 关系中不包含重复的元组。
- 每个关系中的元素是无序的，从上到下进行排列。

---

1 参见第 3 章的脚注，关于“每一个关系”的含义的解释。

- 每个关系中的属性是无序的，从左到右进行排列。
- 关系总是规范化的（例如，第一范式的缩写为 1NF）。

关系中不包含重复元组。这个特点说明了一个逻辑上的事实，即关系被定义为集合，而从数学上来说，集合中不包含重复的元素。

每个关系中的元素是无序的，从上到下排列。这个特点也是一个逻辑上的事实，即关系是一个集合，但在数学上规定集合中的元素是没有次序的（例如，{*a,b,c*} 和 {*c,a,b*}是等价的）。当然，当我们把关系表示成一个表格时（例如，在纸上），我们是按照从上到下的次序来排列这些行的，但是这个次序是任意的，你可以不考虑它。例如，图 2.1 中描述的供应商关系表，表中的行是没有任何次序的。我们可以先描述 S3，然后描述 S1，之后可以依次描述 S5、S4、S2，这和图 2.1 表示的是同一个关系。因此要注意，在关系中没有"第 1 个元组"、"第 5 个元组"、"第 87 个元组"的说法，也不会说"下一个元组"。换句话说，这些元组没有位置的定义，也没有"下一个"的说法。（注意，偶尔我们规定了这样的次序，是因为我们需要一些特定的、附加的运算，例如，"检索第 n 个元组"、"插入一个新元组"、"把这个元组从这儿移到那儿"等等。在第 7 章中我们将会特别介绍这个问题。）

每个关系中的属性是无序的，从左到右排列。同样，关系中的属性也是无序的，从左到右排列，这是因为标题也被定义为集合。当然，当我们把关系表示成一个表格时（例如，在纸上），我们是按照从左到右的次序来排列这些列的，但是这个次序是任意的，你同样可以忽略它。例如图 2.1 中的供应商关系，是按照从左到右的次序排列每一列，即STATUS、SNAME、CITY、SNO，这和图 2.1 表示的是同一种关系。因而，不会有"第 1 个属性"、"第 2 个属性"、"下一个属性"的说法（例如，在属性中不会有"下一个"的定义）。关系的属性通过名字标识，而不是它们的位置[1]。

关系总是规范化的（例如，第一范式的缩写为 1NF）[2]。在非正式的情况下，这里所有语句的含义都是在说，我们所讨论的关系中的每一个元组都是与相应的标题一致的（这一点我们早已经知道了）。所以，一个明显的问题就是关系可以是非规范化的吗？答案是否定的。"非规范化的关系"从术语上来讲就是矛盾的。然而，表格可能真的是非规范化的，这与关系正好相反。这部分内容我将在此书的第三部分详细讲解。

---

1 实际上，不需要我们关心的真正原因是，数学中定义的关系是把属性按照次序从左到右的次序排列的（参见附录 C），并不像关系模型中定义的。

2 你可能知道，有第一范式，就有可能定义更好一级的范式，如第二范式、第三范式等，这与数据库的设计规则有关。在本书的第二部分我将会介绍该内容。

2.2 关系变量

我们再来看看图 2.1，该图给出了 3 个关系，即在某个特定的时间数据库中已经存在了这些关系。但是如果换个不同的时间再来看看这个数据库，我们也许会看到不同的关系。换句话说，这些在图中标识为 S、P、SP 的对象实际上是变量，明确地说是关系变量。像所有的变量一样，它们在不同的时间会具有不同的值。因为按照特性划分，它们是关系变量，所以在任何给定的时间，它们的值也是关系型的数值（如图 2.1 所示，给出的是 3 个关系型的数值）。

现在，因为 S、P、SP 都是真正的变量，它们当然也可以被修改、被赋值。（像第 1 章提到的，变量可以被赋值，也可以把值赋给变量。）例如，假设变量 S 当前具有的值如图 2.1 所示，然后，假设我们执行下面的关系型赋值：

```
S := S WHERE CITY ≠ 'London';
```

和所有的赋值语句一样，上面语句的作用是计算右边表达式的值，然后把结果赋值给左边的目标变量。所以，关系变量 S 的值如下所示：

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

换句话说，供应商 S1 和 S4 的元组（二者的 CITY 属性值都为 London）都被删除了。事实上，前面的赋值语句与下面的 DELETE 语句是等价的：

```
DELETE S WHERE CITY = 'London' ;
```

因而，这个 DELETE 语句是比前面的赋值语句更友好的一种“速记”方法（“速记”加上引号，是因为它实际上要比所期望的缩写长一些）。然而，这也没有关系（不管修改操作是被规范化为关系型的赋值，还是采用更为友好的以这种速记方式来表示），因为从概念上来说都是 S 的旧值已经被新值完全替代了。旧值（带有 5 个元组）和新值（带有 3 个元组）很明显是不一样的，它们具有不同的值。

实际上，现在关系型的 DBMS 都支持作用于关系变量的 INSERT、DELETE 和 UPDATE 运算符，至少从概念上来说，这些运算符都只是特定关系型赋值的一种速记方法。从逻辑上来说，在关系型赋值运算中我们只需要修改运算符（这一点我将在第 4 章中做出说明）。现在我来更多地展示一些速记方法的例子，第一个就是另外一个 DELETE 的例子：

```
DELETE SP WHERE QTY < 150 ;
```

对于这个 DELETE 的结果，我将其作为练习留给读者，样本数据如图 2.1 所示。

下一个例子是 INSERT：

```
INSERT SP RELATION { TUPLE { SNO 'S5' , PNO 'P1' , QTY 250 } ,
                     TUPLE { SNO 'S5' , PNO 'P3' , QTY 450 } } ;
```

INSERT 运算的结果是把一个由 2 个元组组成的关系插入到关系变量 SP 中。假设这个变量的初值就是图 2.1 所示的供应关系，那么执行 INSERT 后，这个关系的基数就变为 14，即 12 个已存在的元组再加上 2 个新的元组。

最后一个例子是 UPDATE：

```
UPDATE SP WHERE SNO = 'S2' : { QTY := 2 * QTY };
```

仍然假设关系变量 SP 的初值为图 2.1 所示的关系，执行 UPDATE 后，会得到一个不同的关系，因为供应商 S2 的 QTY 值翻倍了。顺便提一下，要注意该语句中的属性赋值 QTY := 2 * QTY（要放在大括号内）。

再强调一下，实际上现在关系型的 DBMS 都支持作用于关系变量的 INSERT、DELETE 和 UPDATE 运算符，尽管存在这个事实，但至少从概念上来说，这些运算符都只是特定关系型赋值的一种速记方法。然而，一定要注意，所有这些运算符的运算对象（包括关系型赋值）不是一个关系值，而是一个关系变量。相比之下，我们前面章节中提到的限制和投影这样的运算符是对关系值进行操作的。（当然，限制、投影以及我们将在第 4、5 章讨论的其他运算符都是只读运算符，然而 INSERT、DELETE 和 UPDATE 是更新运算符。对于这两类运算符的差异，请参见第 1 章 ）

总结一下，关系值和关系变量之间是有逻辑上的差异的。为此，接下来我要更加详细地对二者加以区分。当讨论关系值时就使用关系值，讨论关系变量时就使用关系变量。然而，大多数情况，我也会把关系 值（relation value）简写为关系（relation）（就像我们把整型值［integer value］简写为整型［integer］一样）。同时，也会把关系变量（relation variable）简写为**relvar**，例如我说供应商和零件数据库中包含 3 个*relvars*[1]。注意：术语*relvar*没有被广泛使用，但实际是应该被广泛使用的。

最后一点，通常用 R 表示一个关系变量。就像关系一样，R 具有特定标题 H（这是一个别名，具有特定的属性和特定的度）。定义 R 后，标题 H 就是固定的（参见第 3 章），被合法赋值给 R 的每个关系 *r* 必须具有相同的标题 H。而且，在某一特定时间 *t*，把关系 *r* 赋值给 R，那么 *r* 的定义体、元组、基数就是在时间 *t* 的 R

1 更确切地说，是 3 个真正的或者基本的关系变量。这是为了把它们和虚拟的关系变量或视图加以区分（进一步的讨论可以参见第 3 章和第 7 章 。

的定义体、元组和基数。因此，关系变量的定义体、元组和基数是随着时间变化的，而标题、属性和度是不变的。

2.3　练习

判断下面的叙述哪个是正确的？

2.1　关系（以及关系变量）中的元组是无序的。

2.2　关系（以及关系变量）中的属性是无序的。

2.3　关系（以及关系变量）不能有任何未命名的属性。

2.4　关系（以及关系变量）不允许有 2 个或多个属性同名。

2.5　关系（以及关系变量）不包含重复的元组。

2.6　关系（以及关系变量）总是 1NF 的。

2.7　已定义的关系型属性的类型可以是任意复杂的类型。

2.8　关系（以及关系变量）本身是具有类型的。

附加题：

请尽量清晰地给出关系的定义。

2.4　答案

练习 2.1～2.8 都是正确的。具体说明如下：练习 2.1、2.2、2.4、2.5 和 2.6 都可以用本章的知识来解释。练习 2.3 也是毫无疑问的，因为属性就是采用属性名和类型名的偶对定义的，从叙述上来说，一个未命名的属性是矛盾的。现在就剩下练习 2.7 和练习 2.8 了。

我们首先来看一下练习 2.8，“关系（以及关系变量）本身是具有类型的”。就像我刚说的，这句话是正确的。当然，实际上关系就是一个值，而关系变量就是一个变量，从第 1 章中我们知道每个值都是属于某种类型的，每个变量也是属于某种类型的，所以这句话明显就是正确的。但是在本章中对于关系和关系变量类型本身我故意什么都没有说，因为我想把它们放在第 3 章中来讨论，在第 3 章中我将做进一步的解释。

现在来看看练习 2.7，“已定义的关系型属性的类型可以是任意复杂的类型”。同样，我也说过这句话是正确的。我们可以有包含值的、具有属性的关系和关系变量，这些值可以和我们想象的一样复杂。例如，这些值可以是多边形，可以是数组、XML 文档、照片、音频或视频记录等。然而，它也有许多限制：

- 首先，关系模型明显地禁止了关系变量，因此，在属性值为指针类型的数据库中，意味着该数据库中的关系是不允许把指针类型值的属性赋值为元

组的。注意：前面提到的语句是非常松散的，但是它也说明了这种情况的重要性。至于为什么关系模型要强烈禁止，有很多原因。其中之一，Codd在他的书中 *The Relational Model for Database Management Version* 2（Addison-Wesley,1990）给出如下解释：

假设所有用户都能理解比较值的这个动作，却很少有人能理解指针的复杂性……即使恰好有用户理解指针的复杂性，指针操作也要比执行比较的动作更容易造成错误。

换句话说，指针是很复杂的，容易产生错误，从逻辑上来说不是必须的[1]。例如，要查找与给定供应商元组一致的供应关系元组，我们不能使用的是指针类型，但是我们可以通过比较SNO来替代，因为给定SNO值的供应商元组可能存在于供应关系元组中。注意：顺便值得一提的是，在关系模型中对于指针的禁用是隐含的，还有作为关系模型的替代而提出的各种方案也是禁用指针的。

- 第二点，说明起来有点难度，但是可以归结为一点就是，如果关系 *r* 有标题 H，那么在任何级别的嵌套中，*r* 的属性都不能使用具有相同标题 H 的关系类型来定义。这种禁止的目的是排除无限递归的可能性。对于关系类型的详细讨论请参见第 3 章。

至于术语关系的精确定义：首先我们给出术语元组的准确定义，如下：

定义：*T1*，*T2*，…，*Tn*（*n*≥0）表示类型名，与每个 *Ti* 相关的 *Ai* 表示不同的属性名，*n* 个属性名和类型名的组合中的每一个都是属性。与每个属性相关的属性值记为 *Vi*，其类型为 *Ti*；*n* 个属性和值的组合中的每一个都是一个元素。这 *n* 个元素的集合就定义好了，*t* 是一个元组值（或者简称为元组），它具有属性 *A1*，*A2*，…，*An*。*n* 就是 *t* 的度，度为 1 的元组是一元的，度为 2 的元组是二元的，度为 3 的元组是三元的，以此类推，度为 *n* 的元组是 *n* 元的，所有 *n* 个属性的集合就是 *t* 的标题。

现在我们定义关系：

定义：H 表示元组的标题，*t1*，*t2*，…，*tm*（*m*≥0）是不同的元组，都具有标题 H。具有标题 H 和元组集合｛*t1*，*t2*，…，*tm*｝的组合就是关系值（简称为关系），记为 *r*。这些元组的属性为 *A1*，*A2*，…，*An*。*r* 的标题是 H，*r* 具有和标题一样的属性（因此也具有同样的属性名和类型）和度。元组集合｛*t1*，*t2*，…，*tm*｝是 *r* 的定义体，值 *m* 是 *r* 的基数。

---

1 理解我们这里讨论的是关于第 1 章中提到的“逻辑数据库”。相反，物理数据库几乎都要包含指针，而且还很多，但是这些指针对于用户来说是不可见的（用行业术语解释，就是被封装了）。

第 3 章

码、外码和相关概念

保持呼吸！这才是关键（key）。

—— Gimli the Dwarf，在 2002 年的电影 *The Two Towers* 中说的，所以 J. R. R. Tolkien（1954）把它作为了一本书的名字

每个关系变量都有一个码（有时可能是多个），同时有些关系变量还具有外码。这一章主要是解释这些概念并探讨它们的含义。

3.1 完整性约束

每个数据库都具有完整性约束（简称**约束**），这就意味着进行修改操作时要受到限制。下面给出了供应商和零件数据库可能具有的一些约束条件：

- 供应商的状态值（status）必须在 1～100 之间。
- 零件重量必须大于 0。
- 伦敦的供应商的状态值（status）必须为 20。
- 红色零件必须存放在伦敦。
- 状态值（status）小于 20 的供应商不允许供应 P6 号零件。

……

现在，对于每个关系变量都有一种特殊的约束称作码约束。不严格地说，该约束的含义是关系变量的某些属性的组合可以充当那个关系变量元组的唯一的标识符（即码）。例如，在任何给定的时间 *t*，关系变量 S 的每个元组都有一个供应商号码（例如，SNO），它是唯一的，这就意味着在同样的时间 *t*，对于同样的关系变量，它与其他每一个元组的 SNO 值都不相同。因而，在任何给定的时间关系变量 S 的元组都通过供应商号来唯一标识。同样，关系变量 P 都通过零件号进行唯一标识，关系变量 SP 通过供应商号和零件号共同来标识。

顺便提一下，码约束本身就是完整性约束。例如，考虑供应商变量 S，假设供应商号码在这个关系变量中是唯一的，如果插入一个与现有的 SNO 值相同的元组，或者把某个元组的 SNO 值改为一个已经存在的 SNO 值，则都会失败。

我曾经说过，每个关系变量都有码。事实上，这应该是显而易见的事实，原因如下，首先，在给定的时间关系变量的值就是一个关系；其次，关系中不允许包含重复的元组。因此，所有属性的组合（例如，整个标题）肯定包含独一无二的属性。实际上整个标题中只有真子集会具有独一无二的属性，但我们保证整个标题肯定会有唯一的属性，所以才会有码。

集合理论注释：上面段落中使用了集合理论术语真子集（proper subset）。然而并不是所有人都熟知这个术语，所以我岔开话题来解释一下这个术语，假设 A 和 B 都是集合，且 A 是 B 的子集（或者说 A 包含于 B），记作 A⊆B，其含义是 A 中的每一个元素都是 B 中的元素。相反，A 是 B 的真子集，记作 A⊂B，即 A 是 B 的子集，但至少存在一个元素，它在 B 中，而不在 A 中。因此要记住，每个集合都是其自身的子集，但不是真子集。

作为变体，上面的理论完全适用于超集，B 是 A 的超集（或 B 包含 A），记作 B⊂A，与 A 是 B 的子集是同样的含义。同样，B 是 A 的真超集（或 B 真包含 A），记作 B⊃A，与 A 是 B 的真子集相同。因此要记住，每个集合都是其自身的超集，但不是它自身的真超集。

最后，还有几个术语介绍。首先，空集（*empty set*）是不包含任何元素的集合，记作{}或者Ø。空集是任何集合的子集，也是除它自身外，是任何集合的真子集[1]。其次，一个集合含有元素，但也包括子集。对于该部分内容的进一步讨论可以参照附录C。

3.2 码

下面是关系术语码（key）的明确定义[2]：

定义：K 是关系变量 R 的标题的一个子集，当且仅当它满足以下条件时，K 是 R 的码：

1．**唯一性**：在 R 中不可能存在与 K 具有相同值的元组。

---

1 这也是很重要的！但是这不是解释这个的地方。我只能说通常情况在关系型世界中，集合都是唯一的。如果恰好遇到了空集合，它也是没有任何实际意义的。详细解释可以参照本书的附录 B 和 C 以及 Hugh Darwen 的论文 *The Nullologist in Relationland*，包含在 Hugh 和我在 1989～1991 年之间写的一本书 *Relational Database Writings* 中。注意，术语非逻辑学家（*nullologist*）来自于希腊语 nullology，意思是根本什么也不研究（换句话说，就是研究空集）。在 SQL 中没有处理空值的方法，处理空值也是被强烈反对的，我们在本书的第三部分也会看到。

2 为了简化码的概念，有时也把码称为候选码。

2．**不可简化性**：K 的任何真子集都具有唯一性。

如果 K 包含 *n* 个属性，那么 K 的度就为 *n*。

对于该定义，需要强调以下几点：第一，码是属性的集合，本质上不是属性（即使讨论的集合只包含 1 个属性），因为它们都被定义成了相应标题的子集（但不是真子集）。因而，关系变量 S 的唯一码是{SNO}（注意使用大括号），不是 SNO。同样，关系变量 P 的唯一码是{PNO}，关系变量 SP 的唯一码是{SNO，PNO}。顺便说一下，注意{SNAME}不是关系变量 S 的码，因为图 1.1 和图 2.1 中给出的 SNAME 的值不是唯一的。

第二，唯一性是很明显的，但是我要解释一下不可还原性。考虑关系变量 S 和其属性集合{SNO，CITY}（称作 SC），它是标题 S 的子集，具有唯一性（关系变量 S 的有效值中不存在这样的关系，它具有和 SC 相同的值）。但是，它不是不可简化的，因为我们不关心属性 CITY 和剩下的属性。单独的属性集合{SNO}仍然具有唯一性。所以我们不把 SC 看作码，因为它“太大了”。相反，{SNO}是不可简化的，所以它是码。

因此，为什么我们想让码具有不可简化性呢？一个重要的原因就是如果我们明确提出码不具有不可简化性，那么 DBMS 就不能执行恰当的唯一性约束。例如，假设我们告诉 DBMS（这是说谎！），{SNO，CITY}是关系变量 S 的码，那么 DBMS 将不会（事实上，也不能）执行这个约束条件，即供应商号码在全局上是唯一的。相反，它将会，也只能执行较弱的一个约束，即供应商号码在局部上是唯一的。因此，这就是为什么我们需要码，但码中不能包含进行唯一性标识时不需要的属性的原因（但不是唯一的一个原因）。

第三，请注意码的概念适用于关系变量，但不适用于关系。为什么？因为说某个事物是码，就是在说某个完整性约束在起作用，并且完整性约束适用于变量而不是数值。（按照定义，完整性约束中对修改进行了限制，而修改操作作用于变量，不作用于数值。进一步讨论请参见第 6 章 ）

第四，也是最后一点，供应商和零件数据库中的所有关系变量恰好都只有 1 个码。但是一个关系变量可能会有 2 个或多个码。下面是几个这样的例子（只是扼要说明）。

例 1：

```
TAX_BRACKET { LOW ... , HIGH ... , RATE ... }
   KEY { LOW }
   KEY { HIGH }
   KEY { RATE }
```

第 1 个例子的意思是，如果你的应纳税收入在 LOW 和 HIGH 之间，就按照特定的 RATE 交税。例如，如果你的应纳税收入在$10,000～$19,999 之间，那就应该

上交 10%的税；如果在$20,000 至$29,999 之间，则应该上交 15%的税等。假设在一个公正的税收系统中，税率随着应纳税收入的增加而增加，{LOW}、{HIGH}和{RATE}很显然就都是码。

例 2：

```
ROSTER { DAY ... , TIME ... , GATE ... , PILOT ... }
   KEY { DAY , TIME , GATE }
   KEY { DAY , TIME , PILOT }
```

这个关系变量表示简化了的航班时刻表：在某一天的某个时间，某个飞行员驾驶特定的航班从特定的登机口起飞。实际上，因为同一个飞行员不可能在同一天的同一时间在两个不同的登机口起飞，因此{DAY，TIME，GATE}和{DAY，TIME，PILOT}都是码。注意，这个例子中的码有重复的部分。因而，码中有重复部分也是可能的，唯一的原因就是所讨论的问题的码是组合的（就像表示供应关系的关系变量 SP 的唯一码一样）。注意，不是组合而成的码有时也被称作是简单码，就像关系变量 S 中的｛SNO｝和关系变量 P 中的｛PNO｝。

例 3：

```
PLUS { A ... , B ... , C ... }
   KEY { A , B }
   KEY { B , C }
   KEY { C , A }
```

这个例子的含义就是每个元组中的C的值等于A和B的和[1]。显然{A，B}、{B，C}和{C，A}都是码（这些码是复合的，也有重复）。

对于主码的解释：如果关系变量 R 具有一个以上的码（就像上面的几个例子，这是很常见的，但不是必须的），挑选其中之一称作主码。如果关系变量 R 只有一个码，只因为讨论的码是组合的（就像关系变量 SP 中表示供应关系的唯一码）。注意，像关系变量 S 中的码{SNO}和关系变量 P 的码{PNO}，它们不是组合的，有时又称为简单码。

在最后这个例子中，其含义是每个元组的 C 值都等于 A 和 B 的和。显然，{A,B}、{B,C}和{C,A}都是码（码是组合的，但是也有重复 。

关于主码的解释：如果关系变量R有 1 个以上的码，就像上面的这个例子，通常选出其中一个称之为主码（但并不是必须的[2]）。如果关系变量R只有一个码，通常也称之为主码（但这也不是必须的）。然而，除了从关系模型本身的角度来看以外，不管这些码

---

1 实际上，PLUS 可能是关系常量，而不是关系变量。对于关系常量的定义，可以参见 *SQL and Relational Theory*，其中有进一步的讨论。

2 事实上，在关系模型的最初定义中这是必须的，但是现在没有更好的逻辑理由来解释这种必须性。

是否被标识为主码，选择其中哪一个都基本上是心理问题（如果可以有选择的话）。码本身是必要的，但主码不是必要的。然而，在图 1.1 和图 2.1 中，关系变量只有一个码，我就用双下划线标出属性作为主码来进行标识。

3.3 外码

现在我们知道每个关系变量都至少有一个码（受到至少有一个码的约束限制）。但是有些关系变量还要受到一个特定的外码（foreign key）的约束。例如，{SNO}就是关系变量 SP 的外码，要参照关系变量 S 中的主码{SNO}。这就意味着，在给定的时间 *t*，关系变量 SP 包含一个元组，它的 SNO 的值为 S3，那么关系变量 S 也必须在同样的时间 *t* 含有一个元组，它的 SNO 值也为 S3。否则，关系变量 SP 将会展示一些由不存在的供应商所供应的零件，因而这个数据库就不是一个真实情况的正确模型了。

当然，{PNO}也是关系变量 SP 的外码，它的值要参照关系变量 P 的主码{PNO}的值。不严格地说，在某种程度上我们可以认为外码约束就像胶水一样把数据库绑定在一起，话虽如此，这里还是要给出一个定义。

定义：关系变量 R2 和 R1，如果满足以下条件：

1．K 是 R1 的码。

2．FK 是 R2 标题的一个子集。

3．在任何时候，R2 中每个 FK 的值都要与 R1 中某些元组 K 的值相等。

那么 FK 就称为 R2 的外码，它的值要参照 R1 的 K 值。注意，如果 FK 由 *n* 个属性组成，那么 FK 的度就为 *n*。

实际上前面的这个定义是被简化了的，但对于本书要达到的目的来说已经足够了。然而，我还要指出一点，K和FK要由相同的属性组成是不言而喻的。例如，在关系变量S和SP中，关系变量S的主码属性为SNO，类型为CHAR，那么相应的关系变量SP中的外码也必须叫做SNO，属性也必须为CHAR。也就是说（着重指出这一点），K和FK中每个属性都要具有相同的名字和相同的属性，它们要相互匹配。事实上，从规范角度来说，它们必须具有相同的属性。参见本章练习 3.2，可以获得进一步的解释说明[1]。

---

1 如果 FK 中某个属性和 K 中与它相对应的那个属性具有相同的类型，但不具有相同的名字（实际上，这种情况是很少见的，就像 *SQL and Relational Theory* 一书中解释的那样），那么就要使用 RENAME 运算符（该运算符将在第 4 章中描述）来帮忙改名。对于这一点的解释说明可以参照 *SQL and Relational Theory*，也可以参照第 9 章的练习 9.5。

注意：我还要再次强调这个术语。首先，外码是关系模型中保持参照完整性的一种实现方式[1]，即外码的值代表着一种参照性，就像关系变量SP中SNO的值S3要参照关系变量S中相应元组的值一样。实际上，参照完整性规则很简单，即如果元组*t2* 参照了元组*t1*，那么元组*t1* 必须存在。因而，如果想要删除一个供应商元组的想法必定要失败，因为不能留下任何“悬挂着的参照关系”（例如，关系变量SP中对于供应商的参照就不存在了，这就是一个“悬挂着的参照关系”）。

定义中的关系变量 R2 和 R1 可以分别称为参照关系和被参照关系（或者称为目标关系）。由此可以推断，R2 中的元组 *t2* 和 R1 中对应的元组 *t1* 就分别成为参照元组和被参照元组（或目标元组）。如果 FK 是 R2 的外码，那么 R1 中对应的码 K 就是被参照码（或目标码）。

3.4 关系变量定义

现在（也是最后）我们有能力来理解 **Tutorial D** 的关系变量 S、P 和 SP 的定义。下面是关系变量 S 的定义：

```
VAR S BASE
      RELATION
         { SNO     CHAR ,
           SNAME   CHAR ,
           STATUS INTEGER ,
           CITY    CHAR }
      KEY { SNO } ;
```

说明如下：

- 关键词 VAR 指出，这个语句从总体上定义了一个变量（当然，在目前的情况下是一个关系变量）。讨论的变量名称为 S，关键词 BASE 标识定义的变量类型，它代表的是“基本关系变量”，这是关系变量 S 的类型。（不严格地说，基本关系变量就是可以单独存在的变量，不需要依靠其他关系变量来定义。其他类型的关系变量也确实存在，其中最重要的是虚关系变量或称为视图，这将在第 7 章给出简要介绍。然而，在这之前，我们还是把关系变量都简化为基本关系变量。）
- 接下来的 5 行定义了变量的类型（当然，也是关系型的。回顾第 2 章练习 2.8 的答案，指出关系变量和所有的变量一样都要具有一种类型）。关键词 RELATION 表示它只一种关系类型，下面 4 行放在一起说明了对应的标

1 从历史记载来看，我认为参照完整性（*referencial integrity*）这个术语来自于 Codd，他在讨论关系模型的特殊性时定义了这个概念。但是和术语相反，这个概念要早于关系模型。

题。标题中属性的顺序没有规定，因为标题是属性的集合。（就像第 2 章给出的解释，在纸上表示集合时通常用大括号把元素包围起来，元素之间用逗号隔开，并且元素的顺序是没有关系的。因而，{*a*,*b*,*c*,*d*}和{*d*,*a*,*c*,*b*}表示同样的集合。）

- 最后一行定义了关系变量的主码{SNO}。注意，KEY 是关系变量定义的一部分，不是类型说明的一部分。

下面我们讨论一下关系类型的定义，首先关系和关系变量一样都要具有某种类型（就像我们在第 2 章练习 2.8 的答案中看到的一样），就像关系变量 S 定义中表示的一样，**Tutorial D** 中的关系类型采用如下形式定义：

```
RELATION heading
```

Heading 的格式：

```
{ attribute commalist }
```

Attribute 采用如下形式定义：

```
attribute-name type-name
```

实际上，有一个相当充分的理由来解释需要关系类型具有这种特定形式名字的原因，参见第 4 章和第 5 章的的闭包，可以得到更清楚的解释。

下面是关系变量 P 的定义：

```
VAR P BASE
      RELATION
         { PNO     CHAR ,
           PNAME   CHAR ,
           COLOR   CHAR ,
           WEIGHT  RATIONAL ,
           CITY    CHAR }
      KEY { PNO } ;
```

与上面定义唯一不同的地方在于类型 RATIONAL 和属性 WEIGHT。RATIONAL 是 **Tutorial D** 的类型名，在其他语言中称作 REAL。它是系统定义的类型，通常由两个整数的比率值表示（例如：3/8、5/12、-4/3）。

实型值的解释：在十进制定义中，实型值具有如下特性，小数部分由有限的非零数字组成，后面可以接无限个 0，在没有损失的情况下，可以把 0 省略（例如：3/8=0.375000...，2/1=2.000…），或者 小数部分由有限序列的数字组成，后面可以接续另一个有限序列的数字，但是第一个必须是非 0 的，也可以无限循环（例如：5/12= 0.41666...）。

最后给出关系变量 SP 的定义：

```
VAR SP BASE
       RELATION
          { SNO   CHAR ,
```

```
        PNO    CHAR ,
        QTY    INTEGER }
    KEY { SNO , PNO }
    FOREIGN KEY { SNO } REFERENCES S
    FOREIGN KEY { PNO } REFERENCES P ;
```

这里唯一与上面不同的是 FOREIGN 的说明，这是很明显的，不需要解释。

3.5　导入数据库

默认情况下，第一次创建时基本关系变量都是空的（例如，刚定义时），也就是说，它们的初值是相应类型的一个空关系 [1]。为了导入数据库，我们要用关系赋值语句。例如，下面的赋值语句，完成的是给关系变量S赋值，它的值取自图 1.1 和图 2.1 的值。

```
S := RELATION
     { TUPLE { SNO 'S1' , SNAME 'Smith' , STATUS 20 , CITY 'London' } ,
       TUPLE { SNO 'S2' , SNAME 'Jones' , STATUS 10 , CITY 'Paris' } ,
       TUPLE { SNO 'S3' , SNAME 'Blake' , STATUS 30 , CITY 'Paris' } ,
       TUPLE { SNO 'S4' , SNAME 'Clark' , STATUS 20 , CITY 'London' } ,
       TUPLE { SNO 'S5' , SNAME 'Adams' , STATUS 30 , CITY 'Athens' } } ;
```

当然我们也可以采用 INSERT 语句进行赋值（虽然 INSERT 语句要依赖于目标关系变量 S，初始值是否为空，这在赋值语句中没有明确说明），如下：

```
INSERT S RELATION
     { TUPLE { SNO 'S1' , SNAME 'Smith' , STATUS 20 , CITY 'London' } ,
       TUPLE { SNO 'S2' , SNAME 'Jones' , STATUS 10 , CITY 'Paris' } ,
       TUPLE { SNO 'S3' , SNAME 'Blake' , STATUS 30 , CITY 'Paris' } ,
       TUPLE { SNO 'S4' , SNAME 'Clark' , STATUS 20 , CITY 'London' } ,
       TUPLE { SNO 'S5' , SNAME 'Adams' , STATUS 30 , CITY 'Athens' } } ;
```

当然，关系变量P和SP都可以采用这种方式进行赋值。注意：在前面 2 个赋值语句中，给关系变量S赋值的这种表达方式是关系标识符（relation literal）的一种 [2]。因此，正如您所看到的，**Tutorial D**中的关系标识符采用如下形式：

```
RELATION body
```

这里 *body* 采用如下形式定义：

```
{ tuple literal commalist }
```

1　这里应该说明一个非空的初值，但是超出了本书的范围。

2　两点说明：首先，关系标识符是通常构造关系时称作关系选择器调用的一种特殊用法；第二，这里给出的语法 1 中的标识符是经过简化的，但只是轻度简化。第一点将在第 7 章解释，第二点参照本章后面的练习 3.3。

tuple literal 采用如下形式定义：

```
TUPLE { component commalist }
```

component 采用如下形式定义：

```
attribute-name literal
```

当然，在定义体中列出的元组顺序是没有关系的，每个元组标识符的组成部分的顺序也是没有关系的。

回到导入数据库的论述，实际上，除了极小的数据库以外，所有的数据库都很冗长、庞大，所以它不能采用关系赋值语句或者 INSERT 语句进行赋值来进行数据库的导入。实际数据库的导入不是采用手工进行的，而是通常由称为数据库管理员（DBA）的这个具有特殊权利的用户进行导入，它通常采用某种实用程序来实现数据库的导入（“导入实用程序包”）。然而，无论这个导入实用程序真正做了什么，从逻辑上来说它都等价于执行了若干的关系赋值语句。

注意：在实际的数据库中，关系变量也许是被定义的，有时也是被 DBA 导入的（通常在第一次创建数据库的时候）。DBA 负责大量管理和维护工作，但这些都超出了本书的范围。

3.6 数据库系统和程序系统对比

回顾第 1 章图 1.3 给出的代码片段，我从编程的角度来说明相似的一些概念：语句、类型、变量、赋值、标识符、只读运算符（包括比较运算符）。我之前说过这些概念都是和数据库相关的（当然尤指关系数据库），所以下面我们看看他们的相似之处。

- **关系语句**：我们已经看到了关系语句，“INSERT、DELETE、UPDATE 语句和 VARE 语句”具有更好的用户友好性（VAR 通常用来定义变量，特殊的时候用来定义关系变量）。
- **关系类型**：从前面的叙述中我们已经看到了关系和关系变量都具有类型，就像 **Tutorial D** 采用关键词 RELATION 定义的一样，后面紧跟着可用的标题。我们也看到了各种属性类型（如：INTEGER、CHAR、RATIONAL），而且我们已经讨论了一些用户自定义的类型。
- **关系赋值**：看上面关系语句的相关说明（第 1 点）。
- **关系标识符**：“导入数据库”这一部分给出了一个标识符的说明，我们也看到了一些表示数量的标识符（如：‘S1’、‘Smith’、20、‘London’等），以及元组标识符（例如，TUPLE { SNO ‘S1’, SNAME ‘Smith’, STATUS 20,

CITY 'London'}）。

- **关系值**：在任何时候，关系变量 S、P 和 SP 的值就是关系值。当然，由关系标识符说明的也是关系值，如上一点所述。
- **关系型只读运算符**：虽然前面我们已经看到了一些例子，但这一点仍将在第 4 章和第 5 章详细讨论。也可以参照第 1 章的练习 1.1。
- **关系型比较运算符**：将在第 4 章详细讨论。

3.7 练习

3.1 考虑下面的关系类型（从本章的关系变量 P 中抽取的一部分）：

```
RELATION { PNO CHAR , PNAME CHAR , COLOR CHAR ,
                                WEIGHT RATIONAL , CITY CHAR }
```

如果考虑从左到右的属性顺序，可以定义多少种零件的关系变量？

3.2 考虑本章给出的外码定义，我曾指出本章默认了外码 FK 和目标码 K 由完全一致的属性组成，但是这个定义为什么要做这样的假设呢？我的意思是说，这个定义是如何依赖于这个假设的呢？

3.3 在 3.5 节，我说 **Tutorial D** 的关系标识符采用了关键字 RELATION 定义，后面紧跟着由大括号包围起来的一组元组标识符，元组标识符之间用逗号隔开 。虽然我没有说这些元组标识符也要具有相同的属性，这些属性也在标题中明确定义了，还有由关系标识符说明的关系类型，但是假设元组标识符的逗号列表是空的（例如，假设我们用一个标识符来标识一个空的关系），那么相应的关系类型应该如何决定呢？

3.4 如果所讨论问题的供应商当前存在一些供应关系，那么如果删除一个供应商将会发生什么事情呢？

3.5 我曾经说过关系模型禁止使用指针，在不同的表示方式下，外码是不是指针呢？

3.8 答案

3.1 对于第一个属性就有 5 种可能的选择，那么对于这 5 种可能性，第二个属性就有 4 种选择；然后，对于这 4 种可能性，第三个属性有 3 个选择，以此类推。因此，就有 5！=5*4*3*2*1=120 种不同的可能性。（也许这个练习会给您一些暗示，为什么关系模型中不要求关系的属性必须从左到右排列顺序。回顾第 1 章开头的引语！）

3.2 这个答案需要有一些特定的背景知识。首先，我需要强调一点，元组就是集合，是一些组件的集合，每个组件由属性和对应的属性值组成。现在，回顾第1章提到的 *v1* 和 *v2* 的值，当且仅当它们具有相同值的时候才认为它们是相等的（这就暗示了它们肯定具有相同的类型）。例如，整型值 3 等于整型值 3，而不等于其他的整型值（也不等于任何其他类型的值）。同样，两个元组 *t1* 和 *t2* 相等，当且仅当它们具有同样的元组。这就意味着，首先，*t1* 和 *t2* 必须同时具有相同的属性集合，假设为 *A1*，*A2*，…，*An*；其次，对于所有的 i（i=1,2,…,n），每个 *Ai* 的值必须等于 *t2* 中 *Ai* 的值。例如，考虑图 1.1 和图 2.1 中供应商 S1 的元组，这些元组与自身是相等的，但不等于任何其他的元组。

另一点是，因为元组是集合，那么它就有子集。例如，这里给出了图 1.1 和图 2.1 中供应商S1 的许多[1]子集的 2 个（采用**Tutorial D**形式定义）：

```
TUPLE { SNO 'S1' , CITY 'London' }

TUPLE { CITY 'London' }
```

在数学上，子集实际上也是一个集合（参见附录 C），所以在关系模型中，子元组实际上也是一个元组。

我们再返回到外码定义。在本章给出的外码定义中，令 FK 表示外码、K 表示相应的目标码。FK 和 K 都是属性的集合（因为它们每个都是相应标题的子集）。因而，FK 和 K 的值也是元组（也可能是真子元组，但肯定是元组）。例如，图 1.1 和图 2.1 中供应商 S1 元组的码值就是真子元组。

```
TUPLE { SNO 'S1' }
```

现在假设我们要向参照的关系变量中插入一个元组（即定义中的关系变量 R2），为了保证不破坏外码约束，系统就要比较新元组 FK 的值与已存在的被引用关系变量 K 的值是否相等（即定义中的 R1）。但是这些比较都是在元组之间进行的，因而，这个操作进行的有意义的唯一办法是让 FK 和 K 具有相同的类型（我的意思是说，这些比较能否合法的唯一办法是让这个评价的值为 TRUE），这就意味着它们必须包含相同的属性集合。

题外话：前面的讨论指出，元组的每个子集也是元组，而且，由此可以推断，（a）标题的每个子集还是标题；（b）定义体的每个子集仍然是定义体。

3.3 在回答这个问题之前，我要明确指出（就像本章中定义的那样），虽然从数学的角度来说是有一个空集，但是在关系模型中可以有许多不同的空关系。事实上，对于每种可能的关系类型也只有一种空关系。例如，空的供应商关系和空的零

1 实际上有多少个呢？

件关系肯定是不同的，因为虽然他们的定义体相同，但是他们有不同的标题，因此他们属于不同的类型 [1]。

现在回到这个问题本身。显然，如果我们想让一个标识符代表某个特定关系类型中的空关系，那就没有办法从定义体的元组中推断这个关系的类型，因为定义体中没有元组。针对这个原因，**Tutorial D** 的关系标识符的语法与本章前面我所说的是不相同的：

```
RELATION body
```

替换为：

```
RELATION [ heading ] body
```

heading 和 *body* 的语法在本章曾经解释过。但是，通常情况下 *heading* 是可选的（这就是我为什么加了中括号的原因），但即使 *body* 是空的，也必须要说明。

3.4 一种可能性就是当引用完整性被破坏时，这种企图就会失败。一种更复杂的可能性（同时由 **Tutorial D** 和 SQL 支持）是允许有“级联删除规则”的，即在定义 SP 中的外码时要特殊说明其关联性，这样对关系变量 S 执行 DELETE 就会在 SP 中自动执行 DELETE 操作，具有相同供应商号的所有元组就会从 S 中删除。

3.5 不，它们不是。论文 *Inclusion Dependencies and Foreign Keys*（*Database Explorations* 一书的第 13 章，由 Hugh Darwen 和我在 2010 年特拉福德撰写的“第三次声明和相关主题的论述”）列出了这 2 个概念的 14 种逻辑上的差异，这里我只提到一种。指针是有方向性的、是单独的、具有特定目标的，相反，外码的值是规则的数值，因而它们和关系数据库中所有的数值一样，可以被称作“多向关联”。例如，包含 PNO 的值为 P4 的供应关系元组（当然，在外码中也必须有这个值）不只是被关联到零件号为 P4 的零件元组，也被关联到所有其它的供应关系元组和任何其他的所有元组（即数据库中别的地方），这些元组的 PNO 恰好具有 P4 值。

---

1 作为一个题外话，两个不同的关系可以具有相同的定义体，当且仅当它们是空关系时。换句话说，除了这种特殊的情况外，两个关系具有相同的定义体当且仅当它们是相同的关系。

第 4 章

关系运算符 I

这是一个狂欢之夜，我却像一只小狗一样在工作着。

——John Lennon 和 Paul McCartney：A Hard Day's Night

在供应商/零件数据库中已经定义了关系变量（参见第 3 章），并且已经导入数据或完成了初始化（参见第 3 章），我们可以使用它们“真正的工作”了。更准确地说，可以在这些关系变量上执行一些查询（例如，发出检索请求），使用的运算符称之为关系代数（relational algebra），在这一章中，要开始探索这些运算符。警告：令人懊恼的是，这一章相当长，你可能一次只能看一部分。为了在这方面给您提供帮助，我没有把练习和答案完全放在本章最后，而是在本章内容常规的讲解中，相隔一段距离就放置了部分的练习和答案。

4.1 Codd 的原始代数

关系代数由运算符的集合组成（不严格地讲），可以允许我们从“旧”的关系产生“新”的关系。每个运算符都可以将一个或多个关系作为输入，产生另一个关系作为输出。例如，减法（difference）运算符（在 **Tutorial D** 中为 MINUS），使用两个关系作为输入，把一个从另外一个中减掉，产生另外一个关系作为输出。输出另外一个关系，这一点是非常重要的（这就是所谓的关系代数中的闭包），重要的原因就是它允许我们写入嵌套的关系表达式。强调一点，因为每个运算的输出都与输入具有相同的类型，每个运算的输出又可以成为另一个运算的输入。例如，我们执行减法操作，即 *r1* MINUS *r2*，它的运算结果要和关系 *r3* 求并集，这个并集的结果要和关系 *r4* 求交集，等等。

和传统算术做个类比，有可能会帮助您理解。在算术中，我们可以计算两个数字 *a* 和 *b* 的和，得到结果为 *c*，这个输出 *c* 还可以作为输入（因为它们都是数字），所以我们可以把这个输出作为一个乘法操作的输入，这个乘法操作的输出还可以作

为减法操作的输入等等。换句话说，在规则的算术运算符中，数字形成了一个封闭的系统，这就是允许我们写嵌套的算术表达式的原因。同样，关系在关系代数的运算符中也是封闭的，因此，我们可以写嵌套的关系表达式。

现在，可以定义任意数量的运算符，它都适合于这个简单的定义，即一个或多个输入都恰好产生一个输出。下面我简单描述一下最初的运算符是什么样的（即Codd在他早期的论文中定义的）[1]。在本章接下来的几部分中我将详细解释这些运算符，并且在第 5 章还要描述一些其他的运算符。注意：如果你不熟悉这些运算符，可以看本书开篇部分的一些描述，虽然理解起来会有困难，但是不要担心。就像我曾经说过的那样，在接下来的部分，我会通过大量的例子，一步一步地详细讲解。

限制（Restrict）

从特定的关系中返回满足特定条件的元组，构成一个新的关系。例如，我们要求返回供应商关系中 STATUS 的值小于 25 的元组。

投影（Project）

返回一个新的关系，该关系中包含特定关系中的所有元组或子元组，但是有些属性被删除了（就像第 3 章中练习 3.2 的答案说明的一样，子元组仍然是元组，子集仍然是集合）。例如，在供应商关系中进行投影，只保留 SNO 和 CITY 属性（因此，属性 SNAME 和 STATUS 被删除了）。

乘积（Product）

返回一个新关系，该关系中所有元组是两个元组的组合，每个元组分别来自于特定的关系。例如，*r1* 表示供应商关系在 SNO 上的投影，*r2* 表示零件关系在 PNO 上的投影，然后（仍然要借助于闭包特性）我们形成一个 *r1* 和 *r2* 的乘积，结果就是 SNO-PNO 所有可能的组合，SNO 的值是供应商关系中的供应商号，PNO 的值是零件关系中的零件号。注意：这里的“乘法”实际上是笛卡儿（cartesian）乘积，运算符有时就称为笛卡儿（cartesian）乘积。

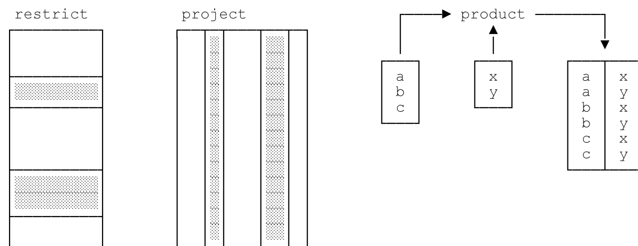


图 4.1　原始的关系代数

1　这里不包括 Codd 附加定义的运算符，即除（divide）。我忽略了这个运算符，因为假设除法解决的问题可以通过 image relations 更好地解决，这部分将在第 5 章介绍。

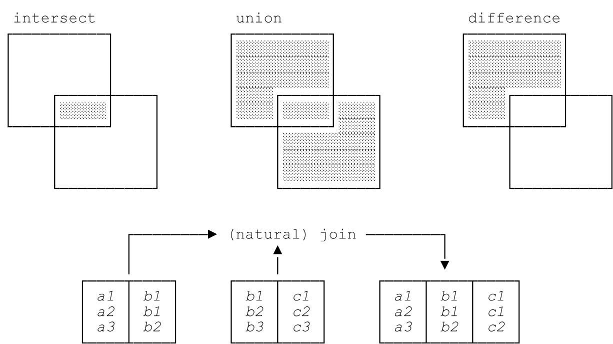


图 4.1 原始的关系代数（续）

交（Intersect）

返回一个新关系，该关系中的所有元组要同时出现在两个特定的关系中。

并（Union）

返回一个新关系，该关系中的所有元组可以出现在两个特定关系的一个当中。

差（Difference）

返回一个新关系，该关系中的所有元组只能出现在第一个关系中，不能出现在第二个关系中。

联接（Join）

返回一个新关系，该关系中的所有元组都是两个元组的组合，两个元组分别来自于两个特定的关系，这两个元组中具有相同的属性，并且具有相同的属性值（这些相同的属性值在结果中只出现一次，而不是两次）。注意：这种联接运算最初叫自然（natural）联接，是为了与其他形式的联接运算加以区分，特别是联接运算（该部分内容将在本书第三部分的第 11 章简单介绍）。因为自然联接是极重要的一种联接运算，非正规的术语联接就成为了一种标准的表示，其含义就是自然联接。本书自始至终都将采用这种表示方式。

最后，结束本部分之前我还要强调几点。

- 代数运算符是通用的：实际上它们可以应用于所有可能的关系中。例如，我们不需要定义一个联接运算符去联接供应商和供应关系，再定义另外一个联接运算符去联接部门和雇员。
- 运算符也是集合，或者说是关系：它们都采用关系作为运算数，而不是单个的元组，也返回整个关系作为结果[1]。

---

1 INSERT、DELETE、UPDATE（和关系复制）都是集合运算符。实际上，关系模型不包括元组级别上的运算符，尽管有时按照元组解释会方便一些（可以参照第 5 章给出的图像关系的例子，或者参见附录 D 的等价描述）。

- 运算符也是只读的：它们读入运算数，返回一个结果，但是不修改任何事情。换句话会所，它们在关系上进行操作，不是在关系变量上进行操作。
- 当然，前面这一点并不意味着关系表达式不引用关系变量。例如，如果 *R1* 和 *R2* 都是关系变量名，*R1* MINUS *R2* 在 **Tutorial D** 肯定是一个有效的关系表达式（只要由这些名字说明的关系变量具有相同的属性，后面详细讲解）。然后，在这种表达式中，*R1* 和 *R2* 本身没有说明这些变量，相反，在对表达式求值时，它们说明了与这些关系变量具有相同值的那些关系。换句话说，我们肯定要使用一个关系变量名把运算数赋予一个关系代数运算符，这样的关系变量引用就构成了一种简单的关系表达式。但是从理论上来说，我们可以使用恰当的关系标识符来替代这个相同的运算数。

做个类比，假设 N 是整型变量，在时间 *t* 的值为 3，那么 N+2 肯定是一个有效的算术表达式，而且在时间 *t* 它正好具有和表达式 3+2 同样的值，不多也不少。

- 最后要说明的是，这些代数运算符也都是只读的，INSERT、DELETE、UPDATE（和关系赋值）本身不是关系代数运算符，虽然它们肯定是关系运算符。虽然有时令人懊恼的是，你可能偶尔遇到与标识相反的一些语句。（当然，实际上 INSERT、DELETE、UPDATE 和关系赋值都是修改运算符，然而我曾经说过，代数运算符都是只读的。）

4.2 限制

现在开始更详细的讨论代数运算符,我们从**限制**（restrict）开始。考虑下面这个查询，查询 1：从零件库中查询零件重量小于 12.5 磅的零件信息。（这里假设，零件重量单位为磅，记录在关系变量 P 中。）下面是 **Tutorial D** 的查询公式：

查询 1：`P WHERE WEIGHT < 12.5`

查询 1 的结果如表 4-1 所示，这个结果是从样本数据中得到的[1]。

表 4-1　　查询 1 的运行结果

PNO	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P5	Cam	Blue	12.0	Paris

正如你所看到的，查询结果是一个关系。而且，结果关系的标题与输入关系的

1 我不会一直这样说了，因为（在第 1 章中已经重复了）我将使用图 1.1 给出的样本数据作为贯穿本书的例子，除非有相反的情况，我会特别说明。

标题是一样的（例如，零件关系就是关系变量 P 的当前值），结果关系的内容由满足限制条件 WEIGHT < 12.5 的特定元组组成。

下面给出**限制**运算符的准确定义。

定义：*r* 是一种关系，*bx* 是 *r* 的限制条件（例如，布尔表达式中每个属性引用都标识了 *r* 的某些属性，而不是任何的关系变量引用）。那么表达式 *r* WHERE *bx* 的意思是要根据限制条件 *bx* 从 *r* 中取值，然后返回一个关系，它和 *r* 具有相同的标题，但是内容由表达式 *bx* 判断为真的那些元组组成。

从这个定义中又引出以下几点：

- 这个定义暗含一个术语限制条件（*restriction condition*），它是一个布尔表达式，每个属性引用都确定与相应关系中的某些属性一致。并且没有关系变量引用。（注意，例子中的布尔表达式真正是使用这个定义给出的限制条件。）就像你已经意识到的那样，布尔表达式的形式如此简单，可以说在某种程度上是因为可以使用相应关系中的特定元组来孤立地判断，而没有必要去检测那个关系中的其他元组，也没有必要去检测其他关系。例如，限制条件 WEIGHT < 12.5 就可以只在零件关系中孤立地判断是真还是假。
- 再重复一次，由 WHERE 子句表示的布尔表达式被假设为一个限制条件。然而，为方便用户，在实际的语言中（包括 **Tutorial D** 和 SQL）通常允许 WHERE 子句包含比这个更通用的布尔表达式，例如本章后面给出的例子。注意：采用这种方式简化约束条件是合法的，因为如果 *exp* 是一个由 *r* WHERE *bx* 组成的表达式，但 *bx* 本身不是一个限制条件，那么 *exp* 就可以等价于某个更为复杂的表达式，在该表达式中涉及的任何限制都确实完全遵守了这个约定。
- 它同时也遵守了这个定义，即限制操作的结果类型总是与输入的类型相同（因为它们都具有相同的标题）。
- 限制有时被称为选择（*selection*），但是后者遭到了轻微的反对，因为它会与 SQL 中的 SELECT 操作混淆（参见本书第三部分），同时也会与关系模型中的选择运算符混淆（参见第 7 章）。

4.3 投影

现在我们讨论投影（project）。考虑下面的查询，查询 2：“对于每个零件，要获得它的零件号、颜色和城市”。这里给出 **Tutorial D** 的定义：

查询 2： `P { PNO , COLOR , CITY }`

查询 2 的运行结果，如表 4-2 所示。

表 4-2 查询 2 的零件库投影运行结果

PNO	COLOR	CITY
P1	Red	London
P2	Green	Paris
P3	Blue	Oslo
P4	Red	London
P5	Blue	Paris
P6	Red	London

正如您所看到，该查询的运算结果仍然是一个关系。它的标题由在投影运算中指定的属性组成，它的内容由相应输入的子元组的值组成。

为了说明另外一个观点，下面给出另外一个例子。查询 3 是“当前零件中颜色和城市的组合是什么？”，这里给出 **Tutorial D** 的定义：

查询 3： `P { COLOR , CITY }`

运行结果，如表 4-3 所示。

表 4-3 查询 3 的运行结果

COLOR	CITY
Red	London
Green	Paris
Blue	Oslo
Blue	Paris

这个例子表明，运行结果的度为 4，不是 6。即使原来的输入中有 6 个元组，但这 6 个元组中的 3 个元组都包含了同样的颜色和城市属性组合。现在，我们想要的结果是一个关系（例如，我们想要保存闭包属性），定义中的关系不包含重复的元组，因而，具有相同颜色和城市属性组合的 3 个元组只保留一个即可。换句话说，“消除重复性”。

题外话：这个例子中最后剩下的属性是“全码”，即唯一的码由整个标题组成（注意图中双下划线的属性）。进一步的讨论请参看第 5 章的练习 5.1。

顺便说一下，前面例子中的消除重复性在直觉上感觉是合情合理的，从逻辑上来说是正确的。再考虑一下原来的查询 “当前零件中颜色和城市的组合是什么？”“red和London”这个组合肯定是存在的，它实际出现了是 3 次，但这个查询没有问组合出现的次数，只问了组合是否存在。所以针对问题“*x*存在吗？”的正确的答案是*yes*或者*no*，而不是数量 [1]。

1 当然，如果我们想要数量的话，也可以询问数量，但这将是另外一个不同的查询，我将在第 5 章讲解如何实现这种查询。

注意：从理论上讲，消除重复性在前面的投影例子中也执行了，但是没有任何影响。没有任何影响的原因是进行投影的属性组合中包含了码（实际上是唯一的码）。超越自己一下，至少我再增加一个概念，即消除重复性。消除重复性一直作为处理代数运算过程的一部分而存在。然而实际上，需要消除重复性的运算符只有投影运算和并运算（将在本章后面介绍）。

下面是投影运算符的定义。

定义：关系 *r* 具有属性 *A1*, *A2*, ..., *An*（也可以采用其他形式表示）。那么表达式 *r*{*A1*，*A2*，...，*An*}的含义是在{*A1*，*A2*，...，*An*}上进行 *r* 的投影，返回的关系具有标题{*A1*，*A2*，...，*An*}，内容包含所有的元组 *t*，元组满足如下条件，即在 *r* 中存在这样的元组，它和 *t* 具有相同的属性值。

对于该定义强调以下几点：

- 当然，在投影操作中属性的说明顺序是没有关系的。这是因为属性名的列表封闭在大括号中[1]。例如，下面这两个表达式从逻辑上来说是等价的，因此是可以相互替代的。

```
P { COLOR , CITY }

P { CITY , COLOR }
```

- 为方便用户，**Tutorial D** 允许在投影操作中采用删除属性的方式来表达，以此来替代保留下来的属性。例如，本章讨论的两个投影，可以采用 **Tutorial D** 的方式表达：

```
P { ALL BUT PNAME , WEIGHT }

P { ALL BUT PNO , PNAME , WEIGHT }
```

注意：**Tutorial D** 允许在整个语言中使用这个类型（不管它在实际上是否有意义）。在 VAR 语句中定义了关系变量 SP，例如，我们用 KEY {ALL BUT QTY} 代替了 KEY {SNO,PNO}。

- 投影运算结果的类型是 RELATION H，标题 H 由执行投影操作的那些属性组成。注意：实际上任何关系操作的结果类型总是标题 H，这里 H 是一个可以应用的标题。因而，我不想更明确地来指明这个事实。

再谈闭包

考虑下面的查询，查询 4：“查询零件关系中零件重量小于 12.5 的零件的号码、颜色和城市？”。这里给出 **Tutorial D** 的定义：

---

1 至少这是正规的语法。在非正规的形式中，通常省略括号，只写出“在 SNO 上的 S 的投影”（替代在{SNO}上的 S 的投影）。

查询 4: `( P WHERE WEIGHT < 12.5 ) { PNO , COLOR , CITY }`
运行结果如表 4-4 所示。

表 4-4 查询 4 的运行结果

PNO	COLOR	CITY
P1	Red	London
P5	Blue	Paris

注意圆括号的用法,它规定了要先判断子表达式 P WHERE WEIGHT < 12.5(它是一个限制条件)。这个限制的结果就是在属性 PNO、COLOR 和 CITY 上投影。因此,这里又一次使用了闭包,限制的结果还是一个关系,所以在其上执行投影是合法的。

这个例子也表明,当我们说(就像定义中写的那样)投影由表达式 *r*{*A1*,*A2*,...,*An*}说明,理解表达式中的符号 *r* 是很重要的。因为表达式本身表示了一个关系表达式,而它通常可以是任意复杂的。当然,在限制 *r* WHERE *bx* 中的符号 *r* 也是同样的。在关系运算符的任何定义中,关系本身的表示都是这样。

4.4 练习 I

4.1 我们可以有正规的理由解释关系中不允许存在重复的元组,你能列举一些实际的原因吗?

4.2 为什么把关系代数称为"关系代数"?

4.3 下面 **Tutorial D** 的表达式的含义是什么?

a. `SP { SNO , PNO , QTY }`

b. `P WHERE CITY = CITY`

c. `P WHERE FALSE`

4.5 答案 I

4.1 正规的理由就是关系的内容被定义为一个集合,数学中的集合不允许有重复的元素。至于实际的原因如下:实际上有很多,但我们不能涉及足够多的基础原理来解释大部分的原因。但至少有一点是很直观的,如下所示[1]:假设"关系变量"**R**允许重复,那么我们就不能说出R的真正的复制品和偶然关系之间的差异(例

1 其他原因将在附录 E 中讨论。

如，因为用户错误而产生的关系）。我们也不能轻易删除这些偶然的关系，即使我们可以把它们识别出来。

还有另外一点可以用来说明关系操作的结果：如果我们不消除这些结果的重复性，那么这些结果就不是关系，我们也不能对它们执行关系操作。换句话说，我们破坏了闭包，我们也不能写出正确的嵌套表达式。

4.2 这是一个很复杂的问题，我认为它不是简单的或者简练的答案[1]。然而，有点不严格地讲，代数可以被定义为一个正规的系统，它包括以下几个部分：（a）某种类型对象的集合；（b）只读运算符的集合，运算符可以作用于这些对象；（c）这些对象和运算符同时满足特定的规则和特性，比如闭包特性（参见附录C）。单词“代数”本身就来源于阿拉伯语*al-jebr*，意思是重置（可能某些东西被破坏了）或组合。关系代数尤其能把前面有限松散的定义组合在一起。

4.3 a.关系变量 SP 的当前值在所有属性上的投影。结果当然是与当前值完全相同的。实际上，其在所有属性上的投影又被称作恒等投影。

b. 这个表达式（当然也是一个限制表达式）返回一个关系，它的值为关系变量 P 的当前值，因为 WHERE 子句中的布尔表达式对每个元组判断后都为 TRUE。实际上，总的来说，限制表达式逻辑上等价于 P WHERE TRUE。注意：逻辑上等价于 *r* WHERE TRUE 的任何限制表达式都被称为恒等限制。

c. 这个表达式（也是一个限制表达式）返回一个空的零件关系。逻辑上等价于 *r* WHERE FALSE 的任何限制表达式都被称为空限制。

4.6 并、交、差

现在开始讨论并（union）、交（intersection）、差（difference）运算，这些运算符都遵循同样的模式，首先从并运算开始。

4.6.1 并

考虑下面的查询，查询 5：“获得至少一个供应商的城市或者一个零件的城市名称”。这里给出 **Tutorial D** 的定义（注意又是一个闭包）：

查询 5： `S { CITY } UNION P { CITY }`

查询 5 的运行结果如表 4-5 所示。

---

1 事实上，关于这个问题我写了一篇论文，即 *Why Is It Called Relational Algebra?* 这篇论文位于我的一本书中 *Logic and Databases: The Roots of Relational Theory*（2007 年，特拉福德）。

表 4-5 查询 5 的运行结果

CITY
Athens
London
Oslo
Paris

正如你所看到的，这个结果仍然是一个关系。它的标题与两个输入关系的标题相同，它的内容包括了这两个关系中的一个或两个（同样要消除重复的元组）。注意：这两个输入关系必须具有相同的标题（等价地，它们必须具有相同的类型），否则将不能进行并操作。但是如果它被定义了，那么结果也应该具有和输入同样的类型。

下面是另外一个例子可以来说明这一点。假设零件具有另外一个属性 STATUS，类型为 INTEGER，考虑查询 “获得至少一个供应商或零件的城市和状态”。这里给出 **Tutorial D** 的定义：

```
S { STATUS , CITY } UNION P { CITY , STATUS }
```

当然，这个例子按照从左到右的顺序说明属性，也是与标题中说明的顺序无关的。

下面给出并运算的定义。

定义：关系 *r1* 和 *r2* 具有同样的类型 *T*，那么表达式 *r1* UNION *r2* 的含义是求出 *r1* 和 *r2* 的并集。它返回一个类型为 *T* 的关系，它的值或者出现在 *r1*，或者出现在 *r2*，或者同时出现在 *r1* 和 *r2* 中的所有元组 *t*。

4.6.2 交

交运算做必要的修正后与并运算非常相似。尤其是这两个输入关系又必须是相同的类型，结果也是相同的类型。下面是一个查询的例子，查询 6：“查询至少包含一个供应商和一个零件的城市”，这里给出 **Tutorial D** 的定义：

查询 6： `S { CITY } INTERSECT P { CITY }`

查询 6 的运行结果如表 4-6 所示。

表 4-6 查询 6 的运行结果

CITY
London
Paris

下面给出交运算的定义。

定义：关系 *r1* 和 *r2* 具有相同的类型 *T*，那么，表达式 *r1* INTERSECT *r2* 的含义就是求出 *r1* 和 *r2* 的交集，返回结果的关系类型为 *T*，内容为同时出现在 *r1* 和 *r2* 中的所有元组 *t*。

4.6.3 差

差运算（在 **Tutorial D** 中为 MINUS）与并运算也有一些相似。特别是这两个运算的输入也要求是同一种类型，输出结果也是同种类型。下面给出一个查询例子，查询 7：“查询至少包含一个供应商，但不包含零件的城市”，这里给出 **Tutorial D** 的定义：

查询 7： `S { CITY } MINUS P { CITY }`

查询 7 的运行结果如表 4-7 所示。

表 4-7　　查询 7 的运行结果

CITY
Athens

然而，差运算与并运算和交运算有一点不同，即差运算要考虑运算数的顺序。在并运算中，*r1* UNION *r2* 和 *r2* UNION *r1* 是等价的。同样在交运算中，*r1* INTERSECT *r2* 和 *r2* INTERSECT *r1* 也是等价的。但是 *r1* MINUS *r2* 与 *r2* MINUS *r1* 表示的不是同一件事情，即使它们返回结果的类型是一样的（返回结果相等当且仅当 *r1* 和 *r2* 相等时，或者二者都是空集）。下面给出一个例子。

查询 8：`P { CITY } MINUS S { CITY }`

查询 8 的运行结果如表 4-8 所示。

表 4-8　　查询 8 的运行结果

CITY
Oslo

下面给出差运算的定义。

定义：关系 *r1* 和 *r2* 具有相同的类型 *T*，那么，表达式 *r1* MINUS *r2* 表示求出 *r1* 和 *r2* 的差（严格按照 *r1* 和 *r2* 的定义顺序），返回结果的关系类型为 *T*，内容为同时出现在 *r1* 但没有出现在 *r2* 中的所有元组 *t*。

4.6.4 一些公式化的特性

并运算尤其具有很多吸引人的公式化的特性，这些特性可以帮助用户完成很多公式化的查询任务[1]。特别说明，该运算符具有交换性、结合性和幂等性。交换性（commutative）的含义就是*r1* UNION *r2* 和*r2* UNION *r1* 是等价的（这个特性遵循这样一个事实，即运算符中*r1* 和*r2* 的定义是对称的）。因此，就像前面已经说明的那样，该运算符中的运算数的顺序是没有规定的。结合性（associative）的含义就是*r1* UNION (*r2* UNION *r3*) 与 (*r1* UNION *r2*) UNION *r3* 是等价的，所以表达式中不需要圆括号，可以简化为*r1* UNION *r2* UNION *r3*。幂等性（idempotent）的含义是*r* UNION *r*与*r*等价。

上面这段论述的特性 100%适用于交运算。也就是说，交运算也具有交换性、结合性、幂等性。但是，这三条特性都不适用于差运算。

4.7 改名

考虑下面这个查询，“至少获得一个与所给名字一致的供应商名或零件名”。（这个查询是人为设计的，也许不能凭借直观的感觉来进行，但是它满足了我要在这里讨论的问题的要求。）下面这个规范化的查询是不能执行的：

```
S { SNAME } UNION P { PNAME }
```

不能执行的原因是并运算（UNION）要求运算数必须具有相同的类型，其含义就是对应的属性不仅要具有相同的类型，也要具有相同的名字。（事实上，规范地说，它们必须具有非常一致的属性。）因此，在目前情况下，为了满足这个需求，执行并运算之前我们必须至少给 SNAME 和 PNAME 之一进行改名（这些属性至少具有相同的类型，但显然它们具有不同的名字）。因此，下面是一个可以执行的规范查询，纯粹是考虑到对称性的原因，我把两个相应的属性都改名为同样的名字。

```
 ( ( S RENAME { SNAME AS NAME } ) { NAME } )
    UNION
( ( P RENAME { PNAME AS NAME } ) { NAME } )
```

在详细解释这个规范的查询如何执行之前（如果你自己还没有弄明白是怎么回事），我们先来讨论一下RENAME运算符本身[2]。下面给出它的定义。

定义：关系 *r* 具有属性 *A*，不具有属性 *B*，那么表达式 *r* RENAME {*A* AS *B*}

---

1 它们也可以帮助优化器，但详细的讨论优化器不是本书的主要目的。

2 你可能已经注意到图 4.1 中没有给出 RENAME，实际上它不是 Codd 的原始运算符之一。这是因为 Codd 从没有正确思考过像 UNION 这样的运算符如何工作的具体细节。

将返回一个与 *r* 具有相同标题的关系，除了将属性 *A* 改名为属性 *B* 之外，内容也与 *r* 相同，除了对属性 *A* 的引用替换为对属性 *B* 的引用之外。

下面给出一个例子。

查询 9：`S RENAME { CITY AS SCITY }`

查询 9 的运行结果如表 4-9 所示（它和通常的供应商关系是一样的，除了将城市属性改名为 SCITY 以外）。

表 4-9　　查询 9 的运行结果

SNO	SNAME	STATUS	SCITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

强调：关系变量 S 在数据库中是没有改变的！表达式 S RENAME {CITY AS SCITY}仅仅是一个表达式（就像 *r1* MINUS *r2* 或 N + 2 都只是表达式一样），像任何表达式一样，它仅仅代表一个值。而且，因为它是表达式，不是语句，所以它可以嵌套在其他的表达式当中。

RENAME只是初步调用JOIN或UNION时才需要[1]。当然JOIN运算的细节也仍然要讨论，但是首先还是要给出一个UNION操作的例子。

```
( ( S RENAME { SNAME AS NAME } ) { NAME } )
  UNION
  ( ( P RENAME { PNAME AS NAME } ) { NAME } )
```

解释：

- S RENAME {SNAME AS NAME}这个子表达式返回一个基本上与关系变量 S 的当前值相同的关系，除了属性 SNAME 被改名为 NAME 以外，我们把结果称之为 *r1*。关系 *r1* 在 NAME 上做投影，获得结果为关系 *r2*。
- P RENAME {PNAME AS NAME}这个子表达式返回一个基本上与关系变量 P 的当前值相同的关系，除了属性 PNAME 被改名为 NAME 以外，我们把结果称之为 *r3*。关系 *r3* 在 NAME 上做投影，获得结果为关系 *r4*。
- 最后，计算表达式 *r2* UNION *r4*，得到的结果关系度为 1，只具有一个属性 NAME（类型为 CHAR），内容为

1 它在联接特定外码时也会需要（参见 *SQL and Relational Theory*，或者第 9 章中练习 9.5 的答案）

```
TUPLE { NAME name }
```

中所有元组的集合。这里的 *name* 表示来自于关系变量 S 的属性 SNAME 的当前值，或者关系变量 P 的属性 PNAME 的当前值，或者二者都有。

4.8　练习 II

4.4　为什么说在交运算或差运算中，消除重复性不算是一个问题？

4.5　关系型的并、交、差运算符都是建立在集合同名运算符的集合理论基础上的，按照你对它们的理解，请给出这些集合运算符精确的定义。

4.6　下面 **Tutorial D** 的表达式的含义是什么？注意：无论是在这里还是在整本书中，你都可以假设 **Tutorial D** 的投影运算在所有的关系运算符中优先级最高。

a. `P { PNO } MINUS ( SP WHERE SNO = 'S2' ) { PNO }`

b. `( S { CITY } INTERSECT P { CITY } ) UNION P { CITY }`

c. `S { SNO } MINUS ( S { SNO } MINUS SP { SNO } )`

d. `SP MINUS SP`

e. `S INTERSECT S`

f. `( S WHERE CITY = 'Paris' ) UNION ( S WHERE STATUS = 20 )`

4.7　写出下列查询的 **Tutorial D** 表达式：

a. 查询所有的供应关系。

b. 查询由供应商 S2 供应的零件号码。

c. 查询状态值（STATUS）范围在 15 至 25 之间的供应商。

d. 查询状态值（STATUS）小于供应商 S2 的供应商号码。

e. 查询所有供应商城市为 Paris 的供应商供应的零件号码。

4.8　自己设计一些查询，并写出这些查询的 **Tutorial D** 表达式。可以依据你自己设计的数据库，或者依据供应商-零件数据库。

4.9　答案 II

4.4　在输入关系中肯定是没有重复元组的，因为输入的就是关系。接下来考虑 *r1* INTERSECT *r2*，这是一个不太严格的定义，其含义是返回 *r1* 中的最大子集 *r*，同时 *r* 也要出现在 *r2* 中，这样显然就消除了重复性。同样，*r1* MINUS *r2* 的含义是返回 *r1* 中的最大子集 *r*，但 *r* 不能出现在 *r2* 中，显然这也消除了重复性。

4.5　***A*** 和 ***B*** 都是集合，那么在集合理论中 ***A*** 和 ***B*** 的并集就是至少出现在 ***A*** 和

B 中之一的所有元素组成的集合。***A*** 和 ***B*** 的交集就是同时出现在 ***A*** 和 ***B*** 中的所有元素组成的集合。***A*** 和 ***B*** 的差集就是出现在 ***A*** 中，但不能出现在 ***B*** 中的所有元素组成的集合。进一步的讨论可以参见附录 C。

4.6 注意：下面给出的表达式的解释说明都是故意采用不规范的写法来进行的。

a. 查询没有由供应商 S2 提供的零件号码。

b. 查询所有零件所在的城市。这个表达式总的来说将零件号限定在 P{CITY}。注意，这个练习说明了**吸收律**（absorption laws），我再强调一下：

```
( r1 INTERSECT r2 ) UNION r2 ≡ r2
( r1 UNION r2 ) INTERSECT r2 ≡ r2
```

符号“≡”表示恒等于。

c. 查询至少提供 1 个零件的供应商号码。

d. 获得供应关系的空集。

e. 查询所有的供应商。

f. 查询满足供应商城市为 Paris 或者状态值为 20 的供应商。

4.7 下面给出的答案不是唯一的：

a. SP。

b. (SP WHERE SNO = 'S2') { PNO }。

c. S WHERE STATUS ⩾ 15 AND STATUS ⩽ 25。这个答案表明，WHERE 子句中的布尔表达式允许使用逻辑运算符 AND、OR 及 NOT。

d.
```
( S WHERE STATUS < STATUS FROM
            ( TUPLE FROM
                  ( S WHERE SNO = 'S2' ) ) ) { SNO }
```

对这个表达式的解释如下，虽然下面的解释有些不规范。第一，这一点以前我们已经说明过，即在 **Tutorial D** 中，WHERE 子句中的布尔表达式可以仅仅是一个简单的限制条件（或者可以更复杂）；第二，嵌套最深处的子表达式 S WHERE SNO='S2'的值为只包含一个元组的关系，但是它仍然是关系，我们需要的不是关系本身，而是包含在这个单个元组关系中的单个元组的状态值（*status*），所以我们首先需要使用 TUPLE FROM 从那个关系中抽取出相应的元组（这是一个运算符，可以从只包含一个元组的关系中抽取出唯一的元组），然后需要使用 STATUS FROM（通常，运算符“*attribute name* FROM *t*”可以完成从元组表达式 *t* 说明的那个元组中抽取出特定的属性值）从那个元组中抽取出相应的状态值（*status*）。这个状态值（用 *x* 标识）就是供应商 S2 的状态值，因此这个表达式就简化为限制表达式 S WHERESTATUS<*x*。

注意：这个练习题还有另外一种答案，在本章 4.12 节“修改运算符说明”中的注释中给出。

e.
```
WITH ( t1 := S WHERE CITY = 'Paris' ,
       t2 := t1 { SNO } ,
       t3 := SP RENAME { PNO AS x } ) :
  ( P WHERE ( t3 WHERE x = PNO ) { SNO } ⊇ t2 ) { PNO }
```

解释如下：

首先，WITH本身不是一个运算符，只是一种句法的小把戏，目的是帮我们给子表达式命名，在某种程度上能够让我们既看到树木，又看到森林。因而，*t1* 是*Paris*的供应商，*t2* 是*Paris*的供应商号码，*t3* 是整个供应关系，但是要把PNO改名为*x*。然后，最后一行是要求求出表达式P WHERE *bx*的值，换句话说，它要从零件关系中挑选出特定的子集，然后将子集在PNO上进行投影，返回希望得到的零件号。至于布尔表达式*bx*，首先它涉及了两个关系之间的比较（分别称之为*r1* 和*r2*）[1]。注意这两个关系的度都为 1，而且，它们都是单个属性，并且相同，都为SNO（因而，它们都具有同样的类型）。关系*r2* 就是*t2*，即*Paris*的供应商号码。至于关系*r1*，其实根本没有引用到*r1*（好像它是一个独立的事情），因为实际上它要依赖于我们要讨论的零件关系的元组。如果我们把零件元组用*p*表示，那么*r1* 实际上就是供应零件*p*的供应商号码，因而，对于每一个零件*p*，*bx*都要判断供应零件*p*的供应商是否是*Paris*供应商的子集。如果*bx*为TRUE，就说明由*Paris*所有供应商提供的零件就是我们想要获得的。

当然，如果不想使用 WITH 也可以。下面 **Tutorial D** 的表达式在逻辑上就等价于前面给出的形式。

```
(P WHERE((SP RENAME{PNO AS x}) WHERE x=PNO) {SNO}⊇
                    ((S WHERE CITY='Paris'){SNO})){PNO}
```

注意：SQL 中支持 WITH 结构，但是它不像 **Tutorial D** 中的结构一样有用。因为（就像本书第三部分介绍的一样）SQL 的这种句法结构不具有易读性，它把一个大的表达式分解为了零散的小的表达式。

4.8 略

4.10 联接

我想讨论的 Codd 原始运算符集合中的下一个运算符就是**联接**（join）。尽管我把它放在了最后，但它还是最重要的一个。首先从一个查询例子开始，查询 10：

1 关系型的比较将在本章后面详细讨论。

“对于每一种供应商关系，查询零件号码、数量和对应的供应商的详细信息”。**Tutorial D** 的定义非常简单：

```
SP JOIN S
```

查询 10 的运行结果如表 4-10 所示。

表 4-10　　　　查询 10 的运行结果

SNO	SNAME	STATUS	CITY	PNO	QTY
S1	Smith	20	London	P1	300
S1	Smith	20	London	P2	200
S1	Smith	20	London	P3	400
S1	Smith	20	London	P4	200
S1	Smith	20	London	P5	100
S1	Smith	20	London	P6	100
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400
S3	Blake	30	Paris	P2	200
S4	Clark	20	London	P2	200
S4	Clark	20	London	P4	300
S4	Clark	20	London	P5	400

解释如下：

下面的定义将会让你更清楚，**Tutorial D** 的联接运算是在具有公共属性的基础上执行的。在目前的情况下，联接运算是在供应商号码的基础上执行的，作为两个运算数关系中的唯一属性的 SNO 是公共的。因而，每个 SP 中的元组 *t1* 的 SNO 都具有值为 *s*，每个 S 中的元组 *t2* 的 SNO 都具有相同的值 *s*，然后得到如表 4-10 所示的结果。（当然，实际上对于任何给定的 *t1*，都恰好有这样的一个 *t2*，因为唯一公共的属性 SNO 构成了 SP 的外码，而且是 S 中的目标码。如果不存在 SNO 值为 S5 的元组 *t1*，那么结果中也不会包含供应商号 S5 的元组。）也要注意的是，公共属性 SNO 在结果中只能出现一次，不能出现两次。

在我给出联接运算本身的定义之前，介绍一下**联接能力**（joinability）的概念。关系*r1* 和*r2* 是可以联接的，当且仅当它们同名属性的类型也是相同的时候，意思就是它们恰好具有非常相同的属性[1]。下面给出联接运算的定义（注意：它揭示了

1 忽略它的名字，联接能力的概念不仅可以应用到联接运算中，也可以应用到其他运算中，将在第 5 章讨论这些。我们也可以采用另外一种方式来定义：关系 r1 和 r2 是可以联接的，当且仅当在进行集合并运算时它们的标题是合法的标题。

标题和元组都是集合，因此可在集合理论运算符中进行操作，比如，并运算）。

定义：关系 *r1* 和 *r2* 是可以联接的，那么表达式 *r1* JOIN *r2* 的含义是进行 *r1* 和 *r2* 的联接运算，它返回一个关系，标题为 *r1* 和 *r2* 标题的并集，内容为 *r1* 中的元组与 *r2* 中的元组求得的并集的所有元组 *t* 的集合。

这里应该提醒一下，我用缩写词联接（join）代替了自然联接。下面是另外一个查询例子，查询 11：`P JOIN S`。

按照定义的形式，该联接将在零件和供应商关系的城市属性上进行联接，CITY 就是 P 和 S 中唯一公共的属性，其运行结果如表 4-11 所示。

表 4-11　　查询 11 的运行结果

SNO	SNAME	STATUS	CITY	PNO	COLOR	WEIGHT
S1	Smith	20	London	P1	Red	12.0
S1	Smith	20	London	P4	Red	14.0
S1	Smith	20	London	P6	Red	19.0
S2	Jones	10	Paris	P2	Green	17.0
S2	Jones	10	Paris	P5	Blue	12.0
S3	Blake	30	Paris	P2	Green	17.0
S3	Blake	30	Paris	P5	Blue	12.0
S4	Clark	20	London	P1	Red	12.0
S4	Clark	20	London	P4	Red	14.0
S4	Clark	20	London	P6	Red	19.0

与并、交运算一样，联接运算也具有交换性、结合性和幂等性。

4.10.1 笛卡儿乘积

通常情况下，笛卡儿乘积（简称为乘积）返回一个关系，该关系包含所有可能的元组，每个元组都由两个元组的联接运算组成，每一个元组都来自于两个特定的关系。现在假设我们想计算关系变量 S 和 P 的乘积。关系变量 S 和 P 都具有属性 CITY，就像表面上看起来的那样，它好像应该具有两个 CITY 属性，但那样的话结果就不是一个关系了。回顾第 2 章，关系中的标题不允许有同一名字的两个不同的属性出现。因此，乘积的输入关系不能有相同的属性（通常这个条件都是能够满足的，因为可以借助于 RENAME 运算符）。

然而，为了不使用 RENAME 运算符，我们来考虑一下下面这个简单的查询，查询 12："查询当前所有的供应商号和零件号"。下面给出 **Tutorial D** 的定义：

```
S { SNO } TIMES P { PNO }
```

表 4-12 中没有列出全部的运行结果（全部运行结果有 30 个元组）。

表 4-12　　查询 12 的部分运行结果

SNO	PNO
S1	P1
S1	P2
…	…
S5	P6

下面给出乘法运算符的定义。

定义：关系 *r1* 和 *r2* 没有公共属性，那么表达式 *r1* TIMES *r2* 表示 *r1* 和 *r2* 的乘积。结果会返回一个关系，该关系的标题是 *r1* 和 *r2* 标题的并集，内容为由 *r1* 中的一个元组和 *r2* 中的一个元组求集合并集后构成的一个新元组 *t* 的集合。

正如你所看到的，这个定义与我给出的联接运算的定义是相同的，除了 *r1* 和 *r2* 是可以联接的这个条件以外，这里需要 *r1* 和 *r2* 不能有公共属性。但是稍等一下，可联接的只是意味着具有相同名字的属性必须具有相同的类型。但是如果没有任何属性具有相同的名字，那么这个条件很容易满足！（如果没有任何属性具有相同的名字，那么肯定不存在具有相同名字的属性，它们不具有相同的类型。）因而，*r1* 和 *r2* 不具有公共属性只是 *r1* 和 *r2* 可联接的一种特殊情况。TIMES 是 JOIN 的一种特殊情况。

下面给出另外一个有趣的例子。“查询不供应零件 PNO 的供应商 SNO 的 SNO-PNO 组合”。这里给出 **Tutorial D** 的定义：

```
( S { SNO } TIMES P { PNO } ) MINUS SP { SNO , PNO }
```

这里用 TIMES 替代了 JOIN，没有任何逻辑上的差异。事实上，在 **Tutorial D** 中支持 TIMES 主要是心理上的原因。

4.10.2　再论交运算

就像你可能已经意识到的那样，交运算也是联接运算的一种特殊情况。特殊情况下，*r1* JOIN *r2* 与 *r1* INTERSECT *r2* 在逻辑上是等价的，当且仅当 *r1* 和 *r2* 具有相同的类型。回顾下面的例子，“查询至少一个供应商与零件在同一城市的城市名”。下面是以前给出的规范定义：

```
S { CITY } INTERSECT P { CITY }
```

这里用 INTERSECT 替代了 JOIN，没有任何逻辑上的差异，事实上，在 **Tutorial D** 中支持 INTERSECT 主要是心理上的原因。

4.10.3　原始运算符

INTERSECT和TIMES可以根据JOIN来进行定义，换句话说，到现在为止我所定义的所有运算符并不都是原始的，不严格地讲，如果一个运算符不能根据其他的运算符定义，那么它就是原始的。一种可能的原始运算符集合就是{限制、投影、联接、并、差}。另一种可能就是用乘积代替联接。注意：这里没有提到改名运算你可能感到很惊讶。事实上，改名运算不是原始的运算，虽然我还没有提供足够的基本原理来解释这是为什么（参见第 5 章关于EXTEND的讨论）。然而，正如您所看到的，在原始的和实用的之间是有差异的，我可能更想要使用改名运算符，即使它不是原始的，而不想要INTERSECT和TIMES[1]。

4.11　关系比较

关系的类型对于这条规则是没有例外的，即“=”比较运算符必须适用于任何类型，也就是说，通过两个具有相同类型 *T* 的关系 *r1* 和 *r2*，我们肯定能够检测出它们是否相等。下面是 **Tutorial D** 的一个例子;

```
S { CITY } = P { CITY }
```

左边的被比较字是在供应商关系上对 CITY 的投影，右边的被比较字是在零件关系上对 CITY 的投影，如果两个投影相等，那么比较运算的返回值为 TRUE，否则返回值为 FALSE。（按照给出的样本数据，返回值为 FALSE。）

比较运算符“≠”、“⊆”（包含于）、“⊂”（真包含于）、“⊇”（包含）、“⊃”（真包含）显然也受到支持。注意：在这些运算符中，“⊆”经常被错误地引用为关系包含运算符。下面给出另外一个例子，它使用了“⊃”。

```
S { SNO } ⊃ SP { SNO }
```

这个表达式的含义要着重解释一下，即查询没有供应零件的供应商（返回值为 TRUE 或 FALSE）。

题外话：这里有一个小小的技术问题，回顾一下，关系代数中的运算符都被假设成了一个封闭的系统，在此意义下，每个运算符的结果都被假设为一个关系。但是关系比较运算符的结果不是一个关系，而是一个布尔值，因而后面的这些运算符

---

1 事实上，有可能定义一种关系代数，它只有两个原始运算符。在 *Databases，Types，and the Relational Model：The Third Manifesto* 一书中定义了这样的关系代数，我们称它为 A，该书出版于 2007 年，作者为 Addison-Wesley、Hugh Darwen 和我。

是否被看作是关系代数的一部分还不是很清楚。目前为止，我们关心的最重要的事情是当我们需要的时候，这些运算符是可以使用的。

注意：每个公共的必须条件都可以在某个给定的关系 *r* 和空关系之间进行“=”比较，换句话说，这是检测 *r* 是否为空的一种方法。它可以简化成如下的定义：

```
IS_EMPTY ( r )
```

如果 *r* 是空的，则返回值为 TRUE，否则返回值为 FALSE。在接下来的章节（特别是第 6 章）中我要多处用到这个比较。它的反向运算符也很有用：

```
IS_NOT_EMPTY ( r )
```

这个表达式逻辑上等价于 NOT (IS_EMPTY(*r*))。

4.12 修改运算符的扩充

本章前面曾经注释过，INSERT、DELETE、UPDATE 都是修改运算符，本身是不属于关系代数的。然而，可以依照代数运算符对它们加以解释（事实上，它们已经被定义过了）。例如，**Tutorial D** 中的 INSERT 语句如下：

```
INSERT R rx ;
```

（这里 R 是一个关系变量名，*rx* 是一个关系表达式，其含义是说明与 R 具有同样类型的一个关系 *r*。）上面这个语句也可以简化为关系赋值：

```
R : = R UNION rx ;
```

比如：

```
INSERT SP RELATION { TUPLE { SNO ‘S5’ , PNO ‘P6’ , QTY 700 } } ;
```

可以实现在供应商关系变量SP中插入一个只包含一个新的元组的关系，或者不严格地但更直观地讲，向SP中插入一个元组（回顾一下，严格地说，在关系模型中是根本不存在元组级的运算符的）。注意：也许你想要了解TUPLE FROM，从某种意义上来说它是属于元组级的（回顾练习 4.7d，就是从只包含一个元组的关系中抽取元组），在扩展数据库的环境下，有时是非常需要这样的运算符的，但它本身不是关系模型的一部分[1]。

现在根据并运算来观察前面的 INSERT 定义，它暗示了要插入一个已经存在的元组的想法会执行成功（即使由 R 和 *rx* 说明的关系是不可联接的，也能执行

---

1 如果你仍然怀疑，这里给出练习 4.7d 的另一个答案，它没有使用 TUPLE FROM：((S{SNO,STATUS} TIMES ((S WHERE SNO = ‘S2’){STATUS} RENAME {STATUS AS *x*})) WHERE STATUS < *x*){SNO}。

INSERT）。（当然它不会插入重复的元组，只要考虑到了那些已经存在的元组，就不会产生任何影响。）为此，**Tutorial D** 又额外支持一个称作 D_INSERT（即不可联接的插入）的运算符，语法如下：

```
D_INSERT R rx ;
```

该语句可以速记为如下形式：

```
R := R D_UNION rx ;
```

D_UNION 代表不可连接的并运算。不可联接的并运算和规则的并运算一行，除了它的运算数关系不需要有公共元组以外（否则会出现例外情况），它仍然使用 D_INSERT 插入一个已存在的元组的企图会失败。

现在转向 DELETE，**Tutorial D** 中的 DELETE 语句定义如下：

```
DELETE R rx ;
```

（这里 R 是一个关系变量名，*rx* 是一个关系表达式，其含义是说明与 R 具有同样类型的一个关系 *r*。）上面这个语句也可以简化为关系赋值：

```
R: := R MINUS rx ;
```

例如，下面的 DELETE 语句：

```
DELETE SP RELATION { TUPLE { SNO 'S1' , PNO 'P1' , QTY 300 } } ;
```

可以实现从供应商关系变量 SP 中删除只包含一个元组的关系。或者不严格地说，它从 SP 中删除了一个元组。注意：这里给出的是 DELETE 最常用的形式。本书中前面的例子都采用了“DELETE *R* WHERE *bx*”形式定义。然而，从技术角度讲，用“DELETE”简化 代替“DELETE *R rx*”，这里 *rx* 反过来可以采用“*R* WHERE *bx*”的形式。

当然，前面的这个定义暗示了删除“一个不存在的元组”的想法会成功的（例如，由 *rx* 说明的一个关系根本不存在于由 R 说明的关系中，但要执行删除操作也会成功）。为此，**Tutorial D** 又额外支持一个称作 I_DELETE（即被包含的删除）的运算符，语法如下：

```
I_DELETE R rx ;
```

上面语句可以速记为：

```
R : = R I_MINUS rx ;
```

I_MINUS 代表被包含的减法运算，表达式 *r1* I_MINUS *r2* 与 *r1* MINUS *r2* 是等价的，除了出现在 *r2* 中的元组在 *r1* 中这个条件以外。换句话说，*r2* 必须包含在 *r1* 中（否则会有例外情况发生）。因此，使用 I_DELETE 删除一个不存在的元素的

方法将会失败。

最后，**Tutorial D** 中的 UPDATE 语句肯定与赋值语句是一致的。然而，执行的细节要比 INSERT 和 DELETE 复杂得多，为此我们将其推迟到第 5 章讲解（参见练习 5.2）。

4.13　练习Ⅲ

4.9　向本章前面给出的注释一样，闭包在关系模式是很重要的，同样原因，在二进制的算术中也是很重要的。然而，在算术中，有一种情况会破坏闭包，即被 0 除。在关系代数中有类似的情况吗？

4.10　**Tutorial D** 下面的表达式的含义是什么？

a.
```
( S { SNO , CITY } JOIN  P { PNO , CITY } ) { PNO , SNO }
```
b.
```
S JOIN SP JOIN P
```
c.
```
( ( S RENAME { CITY AS SC } ) { SC } ) JOIN
 ( ( P RENAME { CITY AS PC } ) { PC } )
```
d.
```
SP { SNO } I_MINUS S { SNO }
```
e.
```
( S WHERE STATUS = 20 ) { CITY } D_UNION
( P WHERE COLOR = 'Blue' ) { CITY }
```

4.11　写出下列查询的 **Tutorial D** 表达式：

a．查询伦敦供应商供应的零件号。

b．查询不是由伦敦供应商供应的零件号。

c．查询供应关系中具有供应关系的供应商号和零件号。

4.12　将自己给出的一些查询描述成 **Tutorial D** 表达式，可以依据自己定义的数据库，或者依据供应商-零件数据库。

4.14　答案Ⅲ

4.9　没有！但是我这里不能解释其原因。因为目前还没有足够的基本原理可以来解释这个问题。（然而，至少我可以说出一条原因，请参见附录 B 中的两个特殊关系 TABLE_DUM 和 TABLE_DEE。）进一步的解释可参见 *SQL and Relational Theory* 一书。

4.10　a．返回在同一城市的零件号和供应商号。注意，整个表达式中的两个内部投影从逻辑上来说是不需要的（当然这也不是错误的）。你认为优化器会忽略它们吗？我们希望它这样做吗？

b. 获得由 SNO–SNAME–STATUS–CITY–PNO–QTY–PNAME–COLOR–WEIGHT 元组构成的新关系，其元组值为供应商和零件属于同一城市的元组。（注意 S JOIN SP 的结果只有 2 个属性，即 PNO 和 CITY，这与关系变量 P 一样 ）。

c．由 SC-PC 构成的元组所组成的关系，但其值要满足供应商在城市 SC，零件在城市 PC（这个例子中的联接运算其实是笛卡儿乘积）。

d．例外情况，除非这个城市不是每个供应商都供应零件，这时 I_MINUS 就变为 MINUS（结果就是供应商号的一个空集）。

e．例外情况，除非这个城市不是状态值为 20 且提供蓝色零件的供应商所在的城市，这时 D_UNION 就变为 UNION。

4.11 a. `( SP JOIN ( S WHERE CITY = 'London' ) ) { PNO }`

b. `P { PNO } MINUS ( SP JOIN ( S WHERE CITY = 'London' ) ) { PNO }`

c.
```
WITH ( t := SP { ALL BUT QTY } ) :
     ( ( t RENAME { PNO AS PX } ) JOIN ( t RENAME { PNO AS PY } ) )
                                                        { PX , PY }
```

注意：这个例子可以采用我们常用的样本数据，假设供应商 S1 供应零件 P1 和 P2，结果中就会存在一对（P1，P2）。但是同样也会存在（P2，P1）、（P1，P1）等。我们要消除这种冗余，因此可以限制结果对要满足 PX<PY。

4.12 略

第 5 章

关系运算符 II

男人的判断……因此附上
小空间里的无限财富。
—— Christopher Marlowe：*The Jew of Malta* (1592)

像第 4 章给出的注释一样，为了满足这个简单的定义“一个或多个关系，除了一个关系以外”，就要定义任意数量的运算符。前面的章节中描述了 Codd 的原始运算符（联接、投影等）。在这一章中，我将要描述许多附加的运算符，这些运算符在发明关系模型后就已经定义了。特别要讨论的有：（a）MATCHING 和 NOT MATCHING；（b）EXTEND；（c）映像关系；（d）聚合、分类汇总，以及其他相关的。

5.1 匹配和非匹配

实际上已经证实，大多数的关系表达式（不是所有的）似乎都需要联接运算符，因为它们的规范操作要求需要一个相关的、但逻辑上有区别的运算符，称为半联接（*semijoin*，在 **Tutorial D** 中称为 MATCHING，即匹配）。下面给出一个例子，查询 1：“查询至少供应了一个零件的供应商的详细信息”，表达式如下：

```
S MATCHING SP
```

查询 1 的运行结果如表 5-1 所示。

表 5-1 查询 1 的运行结果

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London

其结果中包含了至少与一个供应关系匹配的供应商的元组。

下面给出半联接的定义（注意，它似乎在可联接性的定义中出现过，即在第 4 章定义的）。

定义：关系 *r1* 和 *r2* 是可联接的，r1 具有属性 *A1*, *A2*, ... , *An*（只具有这些属性），那么表达式 *r1* MATCHING *r2* 代表了 *r1* 和 *r2* 的半联接运算（注意运算次序）。它返回一个关系，该关系等价于 *r1* 和 *r2* 的联接运算在{*A1*, *A2*, ..., *An*}的投影。

换句话说，表达式 *r1* MATCHING *r2* 返回的关系等价于表达式 *r1* JOIN *r2* 返回的关系，但要在关系 *r1* 的属性上做投影。因此，表达式 S MATCHING SP 可以速写为如下的表达式：

```
( S JOIN SP ) { SNO , SNAME , STATUS , CITY }
```

注意：通常情况下，*r1* MATCHING *r2* 与 *r2* MATCHING *r1* 是不等价的。

在相似情况下，大多数的关系运算符似乎都需要差运算符，因为它们正规的格式要求一个相关的、但逻辑上有区别的运算符，称为半差（*semidifference*，**Tutorial D** 中称为 NOT MATCHING，即不匹配）。下面给出一个例子，查询 2：“查询没有供应零件的所有供应商的详细信息”，表达式如下：

```
S NOT MATCHING SP
```

查询 2 的运行结果如表 5-2 所示。

表 5-2 查询 2 的运行结果

SNO	SNAME	STATUS	CITY
S5	Adams	30	Athens

这个结果包含了与供应关系中所有元组都不匹配的元组。

下面给出半差的定义。

定义：关系 *r1* 和 *r2* 是可联接的，那么表达式 *r1* NOT MATCHING *r2* 代表了 *r1* 和 *r2* 的半差运算（注意运算次序）。它返回一个关系，该关系等价于 *r1* MINUS (*r1* MATCHING *r2*)返回的关系。

举例如下，表达式 S NOT MATCHING SP 可以速记为如下的表达式：

```
S MINUS ( S MATCHING SP )
```

顺便说一下，如果 *r1* 和 *r2* 是不能联接的，但具有同一种类型，NOT MATCHING 会发生什么情况吗？给出一个例子，考虑下面的表达式：

```
S {CITY} NOT MATCHING P {CITY}
```

根据定义，该表达式可以速写为如下形式：

```
S {CITY} MINUS (S {CITY} MATCHING P { CITY } )
```

我确信，你也发现了，后面的这个表达式还可以简写成如下形式：

```
S {CITY} MINUS P {CITY}
```

换句话说，规则的差运算符（即 **Tutorial D** 中的 MINUS）实际上是 NOT MATCHTING 的一种特殊情况。因此，在某种意义上，NOT MATCHING 比 MINUS 更基础、重要。然而，注意这种相似的表示不能应用在 MATCHING 和 JOIN 上。这些运算符中的任何一个都不是另一个的特殊情况。

5.2 扩展

考虑下面的查询，“对于每个零件，查询全部零件的信息，包括以克为单位表示的零件重量”。（回顾一下，在关系变量 P 中零件重量是以磅为单位表示的。）此刻我有个想法，可以让你足以相信，如果只有第 4 章介绍的那些运算符（当然，如果只有图 1.1 中给出的那些关系），那么我们就没有办法来规范地表示这个查询。因此这就是我们需要 EXTEND（即扩展）的原因。下面是使用这个运算符实现以前的一个查询的例子，查询 3（注意，1 磅=454 克 ：

```
EXTEND P : { GMWT := WEIGHT * 454 }
```

查询 3 的运行结果如表 5-3 所示。

表 5-3　　查询 3 的运行结果

PNO	PNAME	COLOR	WEIGHT	CITY	GMWT
P1	Nut	Red	12.0	London	54480
P2	Bolt	Green	17.0	Paris	7718.0
P3	Screw	Blue	17.0	Oslo	7718.0
P4	Screw	Red	14.0	London	6356.0
P5	Cam	Blue	12.0	Paris	5448.0
P6	Cog	Red	19.0	London	8626.0

强调：关系变量 P 在数据库中是没有改变的！表达式 EXTEND P : { GMWT := WEIGHT * 454 }仅仅是一个表达式，像任何表达式一样，它只是表示一个值。而且，因为它是一个表达式，不是语句，所以也可以嵌套在其他的表达式中。

事实上，EXTEND 有两种形式或版本。我们一会儿介绍第二个版本，这里先给出第一个版本的定义。

定义：关系 *r* 不具有属性 *A*，那么表达式 *r* ：{*A* := *exp*}返回一个关系，该关系的标题是在 *r* 的标题中扩展了属性 *A*，其内容为所有元组 *t* 的集合，每个元组 *t*

都扩展了属性 *A* 的值，*A* 值由作用于 *r* 的表达式 *exp* 求得。

现在，我可以说 EXTEND 表达式可以嵌套在其他的表达式中。下面举例说明，查询 4：“查询零件重量大于 7000 克的零件号及其重量（以克为单位）”。表达式如下：

```
( ( EXTEND P : { GMWT := WEIGHT * 454 } )
                        WHERE GMWT > 7000.0 )
                                      { PNO , GMWT }
```

正如你所看到的，这也是投影和约束运算的扩展。查询 4 的运行结果如表 5-4 所示。

表 5-4　　　　查询 4 的运行结果

PNO	WEIGHT
P2	7718.0
P3	7718.0
P6	8626.0

实际上，EXTEND 真的是一个很重要的运算符。为此，我要多举几个例子，并给出相应的结果。

1. 查询 5：`EXTEND S : { TAG := 'Supplier' }`

查询 5 的运行结果如表 5-5 所示。

表 5-5　　　　查询 5 的运行结果

SNO	SNAME	STATUS	CITY	TAG
S1	Smith	20	London	Supplier
S2	Jones	10	Paris	Supplier
S3	Blake	30	Paris	Supplier
S4	Clark	20	London	Supplier
S5	Adams	30	Athens	Supplier

2. 查询 6：`EXTEND ( P JOIN SP ) : { SHIPWT := WEIGHT * QTY }`

查询 6 的运行结果如表 5-6 所示。

表 5-6　　　　查询 6 的运行结果

PNO	……	WEIGHT	……	SNO	QTY	SHIPWT
P1	……	12.0	……	S1	300	3600.0
……	……	……	……	……	……	……

3. 查询 7：`EXTEND S : { SCITY := CITY } ) { ALL BUT CITY }`

查询 7 的运行结果如表 5-7 所示。

表 5-7 查询 7 的运行结果

SNO	SNAME	STATUS	SCITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

注意：最后的一个例子（即例子 3）说明一点，就像第 4 章提到的一样，RENAME 不是原始的运算符，因为给出的表达式（投影的扩展）与下面的改名表达式逻辑上是等价的。

```
S RENAME { CITY AS SCITY }
```

现在转向 EXTEND 的第二种形式，这种形式的主要目的是完成“要是…又怎么样？”的查询。换句话说，它用来观察特定变化对数据库产生的影响，但实际上没有产生任何变化。下面给出一个例子（如果零件重量以克为单位来代替以磅为单位，会发生什么？）：

```
EXTEND P : { WEIGHT := WEIGHT * 454 }
```

这个例子还是很简单的，因为圆括号属性赋值中的目标属性不是一个新的属性，而且这个例子在 EXTEND 以前的例子中也看过，它也是特定关系中已存在的属性（在当前这个例子中，该关系就是关系变量 P 的当前值）。整个表达式完全等价于下面这个式子：

```
( ( EXTEND P : { GMWT := WEIGHT * 454 } ) { ALL BUT WEIGHT } )
                                         RENAME { GMWT AS WEIGHT }
```

下面给出 EXTEND 第二种形式的定义。

定义：关系 *r* 具有属性 *A*，那么表达式 EXTEND *r* ：{*A* := *exp*}返回一个关系，该关系的标题与 *r* 的标题相同，内容是所有元组 *t* 的集合，元组 *t* 的值来源于元组 *r*，但元组 *r* 中属性 *A* 的值要用表达式 *exp* 的值替代。

5.3 映像关系

不严格地说，映像关系（image relation）具有某些元组的某个关系的“映像”（通常情况下，也可能是其他关系中的元组，但不是必须的）。例如，还是参照供应商-关系数据库，下面的例子就是供应关系的一个映像（其取值来自于关系变量 SP

的当前值），是对供应商 S4 的取值形成的一个映像，如表 5-8 所示。

表 5-8 供应商 S4 在供应关系 SP 中的映像

PNO	QTY
P2	200
P4	300
P5	400

显然，这个特殊的映像关系可以通过下面 **Tutorial D** 的表达式获得：

```
( SP WHERE SNO = 'S4' ) { ALL BUT SNO }
```

但通常情况下，映像是一个非常有用的、而且应用很广的一个概念，因此我们希望能够有一种简化的方式来定义它。上面这个例子的速记方式如下：

```
S WHERE ( ‼SP ) { PNO } = P { PNO }
```

在这个表达式中，子表达式 ‼SP 是一个映像关系的引用，它表示与 S 的当前元组值一致的一个映像关系。解释如下。

- 首先，整个表达式要求一个关系的特定子集，该关系的值为关系变量 S 的当前值（例如，供应商关系的一个子集）。
- 问题中的子集由所有供应商关系的元组组成，但它的值由 WHERE 子句中的布尔表达式判断为 TRUE 的值组成。这个布尔表达式不是一个简单的约束表达式，它涉及对属性的引用，但引用的属性不是关系变量 S 的属性，它也涉及关系变量的引用。事实上，正如你看到的，它是一个关系型的等值比较。
- 等值比较中右边的运算数是关系变量 P 当前值在 PNO 上的投影，因而包含了 6 个零件号，即 P1，P2，…，P6（或者是包含 6 个零件号的元组）。左边的运算数也是某个关系在 PNO 上的投影，因此运算数关系必须要具有相同的类型。
- 我们可以想象一下，这个等值比较可以按照任意的顺序，对供应商关系中的每个元组依次执行一个元组的等值比较。考虑这样的一个元组，即供应商关系的一个元组 *Sx*（我们可以称之为元组 *t*）。对于元组 *t*，表达式‼SP 的含义是关系变量 SP 的一个当前值关系的相应的映像关系。换句话说，它表示了关系变量 SP 中由供应商 *Sx* 提供的零件组成的 PNO-QTY 构成的集合。例如，如果 *Sx* 值为 S4，它就表示了本部分开头给出的那个关系。
- 对于元组 *t*，表达式(‼SP){PNO}（例如，在映像关系中对 PNO 的投影）表示了由供应商 *Sx* 供应零件的编号的集合。

- 因而，这个表达式的含义就是供应关系 S 中的供应商的集合，这些供应商提供的零件集合要等价于关系变量 P 在 PNO 上投影形成的所有零件编号的集合。整个表达式的含义就是完成查询 8：“查询供应了所有零件的供应商”，其运行结果如表 5-9 所示。

表 5-9　　查询 8 的运行结果

SNO	SNAME	STATUS	CITY
S1	Smith	20	London

进行讲解之前，我要强调一点，“查询供应了所有零件的供应商”并不是一个简单的查询。在 SQL 中，要编写好几行的相当复杂的代码来实现此查询，我们将在本书的第三部分看到这些内容。为了把它规范化成“一行”，可以通过构建一个映像关系来实现，因而映像关系是非常实用的。

顺便再举一个例子，回顾第 4 章中练习 4.7 的 e，实现“查询由巴黎供应商供应的所有零件号码”。在第 4 章中给出的练习答案如下：

```
( P WHERE
    ( ( SP RENAME { PNO AS x } )
           WHERE x = PNO ) { SNO } ⊇
             ( ( S WHERE CITY = 'Paris' ) { SNO } ) ) { PNO }
```

但是它也可以用映像关系实现：

```
( P WHERE ( ‼SP ) { SNO }⊇ ( S WHERE CITY = 'Paris' ) { SNO } ) { PNO }
```

下面给出另一个例子，假设给出一个供应商-关系数据库的修订版本（与我们通常使用的例子相比，它同时被扩充也被简化了），形式如下（只是简要给出）：

```
S { SNO }          /* 供应商 */
SP { SNO , PNO } /* 供应商及供应的零件 */
PJ { PNO , JNO } /* 被某个工程使用的零件 */
J { JNO }          /* 工程号*/
```

关系变量 J 代表工程（*projects*，JNO 代表工程号），关系变量 PJ 代表了工程与零件之间的关系。现在考虑查询，“查询所有的供应商-工程号对（*sno-jno*），但要满足 *sno* 是关系变量 S 中的值，*jno* 是关系变量 J 中的值，并且工程 *jno* 使用了 *sno* 供应的所有零件”。这是一个相当复杂的查询，但是用映像关系表示就很简单了：

```
( S TIMES J ) WHERE ‼PJ ⊆ ‼SP
```

下一部分我们会进一步看到涉及映像关系的例子，但是现在我们用一个定义来

结束这一部分（给出这个定义的目的主要是记录，如果第一次阅读时感觉理解上有点困难，请不要担心）。

定义：关系 *r1* 和 *r2* 是可联接的，它们的公共属性被称作 *A1*, *A2*,…, *An*。*t1* 是 *r1* 的元组，*t2* 是 *r2* 的元组，它们在属性 *A1*, *A2*, …, *An* 上具有相同的值，关系 *r3* 是 *r2* 的约束，它的元组中不包含 *t2*，关系 *r4* 是 *r3* 的投影，但要除去属性 *A1*, *A2*, …, *An*，那么 *r4* 就是与 *t1* 一致的映像关系。

5.4　聚集和分类汇总

关系模型中的聚集运算符（aggregate operator）通常不是关系运算符，因为它的结果不是一个关系。相反，聚集运算符只产生一个单独的值，是某些关系中一些属性值的聚集（聚集可以是一个集合或者包 [1]，否则，在使用COUNT运算符的情况下，还是有点特殊，它的值来自于对整个关系的聚集。下面是几个例子：

```
X := COUNT ( S ) ;                  /* X = 5 */
Y := COUNT ( S { STATUS } ) ;       /* Y = 3 */
Z := SUM ( SP { QTY } ) ;           /* Z = 1000 */
```

就像注释中指出的那样，第一个赋值语句把值 5 赋给了变量 X（即供应商关系的度），第二个赋值语句把值 3 赋给了变量 Y（供应商关系在 STATUS 上进行投影后的度），第三个赋值语句把值 1000 赋给了变量 Z（即供应关系在 QTY 上投影后求得的总和。）通常情况下，**Tutorial D** 的聚集运算符采用如下形式调用：

```
agg ( rel exp [, exp ] )
```

可用的运算符包括：COUNT、SUM、AVG、MAX、MIN、AND、OR、XOR（最后三个应用在布尔值的聚集中）。在 *exp* 中，对于属性的引用可以出现在任何允许使用标识符的地方，中括号的意思是表达式 *exp* 是可选的；在 COUNT 中则必须要省略它，只要关系表达式 *rel exp* 表示的关系的度为 1，那么就省略。在这种情况下，假设了表达式是由关系中只包含一个属性的引用组成。下面给出一些例子：

1. `SUM ( SP , QTY )`

这个表达式的含义是对关系变量 SP 中的所有重量求和（也可以说是对 SP 中重量的所有当前值求和），其结果为 3100。

1　包也称为复合集合，即集合中允许有重复的元素。

2. `SUM ( SP { QTY } )`

这个表达式是SUM(SP{QTY},QTY)的一种速写形式，表示了SP中所有不重复的重量的和，其结果为1000（可以与前面的例子对比）。

3. `AVG ( SP , 3 * QTY )`

这个表达式实现的是对SP中每个供应关系的重量值都翻3倍后求其平均值(答案是775)。

聚集运算的基本思想就是这样。它们主要用在以下两个方面：

- 它们可以用在分类汇总（summarization）运算中。有点不严格地讲，对每个元组进行聚集运算就相当于在某个关系上的投影。
- 它们也可以用在 WHERE 子句中，因此支持对广义约束（generalized restriction）的引用（相当草率地讲，纯粹是为了当前的目的）。

值得提及的一点是，偶尔两种情况可能都会出现，影像关系也几乎都会涉及这两种情况。

5.4.1 分类汇总

考虑下面的查询，查询 9:“对每个供应商，查询供应商号及其供应的零件数量”。下面给出 **Tutorial D** 的规范语句：

```
EXTEND S { SNO } : { PCT := COUNT ( ‼SP ) }
```

查询9的运行结果如表5-10所示（尤其注意供应商S5的元组)。

表 5-10　　查询 9 的运行结果

SNO	PCT
S1	6
S2	2
S3	1
S4	3
S5	0

观察一下，前面这个例子的结果真正构成了一个分类汇总（检查早期给出的不严格的定义)：它由聚集构成（实际上是分类统计)，该聚集作用于供应关系的每个元组上，而每个元组是在供应商关系上对SNO投影后形成的。

顺便再举一个例子，考虑查询10:“对于每个供应商，查询它的供应商号及供应两件的总重量”。**Tutorial D** 的规范定义如下：

```
EXTEND S { SNO } : { TOTQ := SUM (‼SP , QTY ) }
```

查询10的运行结果如表5-11所示（尤其注意供应商S5的元组)。

表 5-11 查询 10 的运行结果

SNO	TOTQ
S1	1300
S2	700
S3	200
S4	900
S5	0

下面看第 3 个，也是最后一个例子，考虑查询 11："对于每个供应商，查询所有供应商的详细信息，以及相应的供应零件的总重量、最大重量和最小重量"。这里给出 **Tutorial D** 的冗长定义（但是它需要进行一些修正，一会儿我将解释这个）。

```
EXTEND ( S WHERE CITY = 'Athens' ) : {  TOTQ := SUM ( ‼SP , QTY ) ,
                                        MAXQ := MAX ( ‼SP , QTY ) ,
                                        MINQ := MIN ( ‼SP , QTY ) }
```

这个例子说明了多重 EXTEND 的用法。注意：我顺便提一下，很多其他的 **Tutorial D** 运算符也存在多重运算的形式，尤其包括 RENAME 和 UPDATE，其他的运算符不在本书的讨论范围之内。

这个例子引出了一个更重要的问题，即如果聚集运算符（即 *agg*）的运算结果是空集，那么会发生什么？我们已经看到，如果 *agg* 是 COUNT 或 SUM，运算结果是 0。至于其他的运算符，如果 *agg* 是 AND，则结果为 TRUE；如果 *agg* 是 OR 或 XOR，则结果为 FALSE；如果 *agg* 是 AVG、MAX 或 MIN，则会例外，在 AVG 情况下，因为如果结果是空集，就意味着对空集求平均值，会出现被零除的情况，在 MAX 和 MIN 的情况下，因为运算符的结果显然是输入的一部分，如果输入是空集就意味着没有值存在。注意：COUNT 和 SUM 返回值为 0、AND 返回值为 TRUE、OR 或 XOR 返回值为 FLASE 的原因是，每一种情况下指示的结果都有相应认定的值。参考 *SQL and Relational Theory* 可以获得进一步的解释。

那么，如果 AVG、MAX 和 MIN 在有可能作用于空集的情况下，我们应该如何处理呢？下面就是前面那个例子的修订版本，它说明了可以使用它们的一些诀窍。

查询 11：`EXTEND ( S WHERE CITY = 'Athens' ) :`

```
{ TOTQ := SUM ( ‼SP , QTY ) ,
  MAXQ :=
  CASE WHEN IS_EMPTY ( ‼SP ) THEN 0 ELSE MAX ( ‼SP , QTY ) END CASE ,
  MINQ :=
  CASE WHEN IS_EMPTY ( ‼SP ) THEN 0 ELSE MIN ( ‼SP , QTY ) END CASE }
```

查询 11 的运行结果如表 5-12 所示。

表 5-12　　　　查询 11 的运行结果

SNO	SNAME	STATUS	CITY	TOTQ	MAXQ	MINQ
S5	Adams	30	Athens	0	0	0

如果有人发现了比前面这个例子更简洁的答案，都可以公开讨论一下。

对 MAX 和 MIN 的说明：事实上，曾经有过争论说如果 MAX 和 MIN 作用于空集，这个定义是根本不存在的。为了确定这一点，我们特别研究一下 MAX（下面的讨论同样完全适用于 MIN）。首先，我们定义一个有双重价值的运算符 LARGER，它会返回两个讨论值中较大的一个，那么，（a）MAX 调用基本上可以替换为 LARGER 调用；（b）LARGER 也有一个等同的形式，即“负无穷大”（含义就是相应类型的最小值）。所以，我们可以有理由定义，即作用于空集上的 MAX 也有一个等同的值。实际上，也许最好的办法就是同时提供两个版本的 MAX（别玩了，它们是不同的运算符），然后让用户自己决定。我们甚至可以提供第三个版本，即可以采用附加的变量 *x*，这里 *x* 表示如果聚集作用于空集，则返回值为 *x*。但进一步的讨论已经超出了本书的范围。

5.4.2 明确的分类汇总

正如你看到的，分类汇总可以采用 **Tutorial D** 中 EXTEND 和映像关系来表示。然而，当前也支持明确的 SUMMARIZE 运算符。下面给出几个本部分前面讨论的几个分类汇总的例子，我们用 SUMMARIZE 来实现。第一个查询是“对于每个供应商，查询供应商号及它所供应的零件数量”，表达式为：

```
SUMMARIZE SP PER ( S { SNO } ) : { PCT := COUNT ( ) }
```

第二个查询是“对于每个供应商，查询供应商及其供应的零件总重量”，表达式为：

```
SUMMARIZE SP PER ( S { SNO } ) : { TOTQ := SUM ( QTY ) }
```

第三个查询是“对于每一个雅典供应商，查询供应商的详细信息和供应零件的总重量、最大重量和最小重量”，表达式为：

```
SUMMARIZE SP PER ( ( S WHERE CITY = 'Athens' ) { SNO } ) :
                                        { TOTQ := SUM ( QTY ) ,
                                          MAXQ := MAX ( QTY ) ,
                                          MINQ := MIN ( QTY ) }
```

然而，按照我的意见，SUMMARIZE 可以从语言中删掉。一个理由是，SUMMARIZE 的表示方式比 EXTEND 要笨拙、更难理解；另一个理由是它使所用于空集的问题更加复杂化了（我敢肯定你已经注意到了，但是我不想给出第 3 个例子中作用于空集的特殊情况）。还有更可怕的一点是，它们要依赖一个已经收到怀

疑的语言结构，即 *summary*（比如例子中的 COUNT（），SUM（QTY）和 MAX（QTY））。注意：我把这个结构称作“受怀疑的”，是因为已经证实它很难跟得上相应的语法的准确定义（一种批判的声音就是它不能应用于映像关系引用中）。在本书中不对此做详细讨论。我只是强调一下，分类汇总是与聚集运算符调用是不一样的。例如，下面的这个例子肯定是非法的：

```
Z := SUM ( QTY ) ; /* 警告! 非法! */
```

正是因为以上的这些考虑，在本书中对 SUMMARIZE 运算符本身不做过多的讨论。

5.4.3 广义约束

像前面我提到的那样，广义约束（Generalized Restriction）肯定不是一个“官方的”术语（在某些情况下，它甚至不是一个很好的术语），但是我发现，在该部分用它在我想讨论的地方做个标签还是很方便的，例如，形如 *r* WHERE *bx* 的表达式，布尔表达式 *bx* 就涉及了聚集运算符的调用，甚至涉及映像关系的引用。下面给出一个例子，即“查询供应关系小于 3 的供应商”：

```
S WHERE COUNT ( ‼SP ) < 3
```

其结果包含了供应商 S2、S3、S5 的元组。

再给出另外一个例子，即“查询供应零件重量小于 1000 的供应商”：

```
S WHERE SUM ( ‼SP , QTY ) < 1000
```

这样的例子还有很多，例如，“查询供应数量大于 250 的所有供应商号及其供应关系”：

```
( EXTEND S { SNO } : { TOTQ := SUM ( ‼SP , QTY ) } ) WHERE TOTQ > 250
```

再比如：修改所有供应零件重量小于 1000 的供应商状态值，把它们的状态值减半。

```
UPDATE S WHERE SUM ( ‼SP , QTY ) < 1000 : { STATUS := 0.5 * STATUS }
```

5.5 练习

5.1 在第 3 章中，在关系的表格中，我用双下划线来表示哪个属性是主码，而且在那一章我也解释了码、主码，但它们都应用于关系变量而不是关系。在这一章和第 4 章中，我也给出了很多表格来代表关系本身（我的意思是说，对于某些关系变量，关系中的数据可能与样本数据不同），在这些表格中我也采用了这种惯用

表示。请您来解释一下这种惯用的表示方法？

5.2 考虑下面 **Tutorial D** 的 UPDATE 语句：

```
UPDATE S WHERE CITY = 'Paris' : { STATUS := 30 }
```

请您给出一个与该语句等价的关系赋值语句。提示：可以采用 EXTEND 的“要是……又怎么样？”版本。

5.3 下面 **Tutorial D** 的表达式是在本章讨论的，位于 5.3 小节“映像关系”部分：

```
S WHERE ( ‼SP ) { PNO } = P { PNO }
```

请您给出一个与映像关系等价的关系表达式，该映像关系引用在 WHERE 子句中（“‼SP”），但不能使用映像关系引用本身。

5.4 下面的 **Tutorial D** 表达式的含义是什么？

a. `P MATCHING S`

b. `S NOT MATCHING ( SP WHERE PNO = 'P2' )`

c. `EXTEND P : { SCT := COUNT ( ‼SP ) }`

d. `P WHERE SUM ( ‼SP , QTY ) < 500`

5.5 采用 **Tutorial D** 表达式完成下面的查询：

a. 查询供应商号码，该供应商所在城市是所给城市字母列表中的第一个（假定至少存在一个供应商）。

b. 查询至少存在两个供应商的城市名称。

5.6 任意给出一个“广义约束”，它可以采用 EXTEND 和标准的约束来表示。

5.6 答案

5.1 要考虑两种情况：（a）描述的关系是某个关系变量 *R* 的样本值；（b）描述的关系可以是某个关系表达式 *rx* 的样本值，这里 *rx* 是不同于简单关系变量应用的表达式（回顾，关系变量引用基本上只是相应的关系变量名）。在第一种情况下，双下划线只是简单指示出主码 *PK* 在 *R* 中已经声明，相应的属性是 *PK* 的一部分。在第二种情况下，你可以把 *rx* 看作是为某些临时关系变量 *R* 定义的表达式（例如，考虑带有 WITH 的特殊情况），那么双下划线可以表示主码 *PK* 在理论上说是为关系变量 *R* 声明的，其属性是 *PK* 的一部分。

5.2 一种可能的答案如下：

```
S := ( S WHERE CITY ≠'Paris' )
      UNION
     ( EXTEND S WHERE CITY = 'Paris' : { STATUS := 30 } ) ;
```

更好一些的答案可以表示如下：

```
S := ( S WHERE CITY ≠'Paris' OR STATUS = 30 )
      UNION
     ( EXTEND S WHERE CITY = 'Paris' AND STATUS ≠ 30 ) :
                                          { STATUS := 30 } ) ;
```

附加练习：在什么情况下可以说后面的答案要“略胜一筹”？

5.3 首先，给出整个初始表达式的一种替代方式是很方便的：

```
S WHERE
   ( SP MATCHING RELATION { TUPLE { SNO SNO } } ) { PNO } = P { PNO }
```

解释如下：考虑 S 的某个元组，假设供应商为 *Sx*。那么针对这个元组，表达式 TUPLE{SNO SNO}的含义是只包含 SNO 值为 *Sx* 的元组(第一个 SNO 是属性名，第二个 SNO 是关系变量 S 中供应商 *Sx* 属性名的值)，所以，表达式如下：

```
RELATION { TUPLE { SNO SNO } }
```

它表示了只包含那个元组的一个关系(事实上这是一个关系选择器调用，参见第 7 章)，因此，表达式如下：

```
SP MATCHING RELATION { TUPLE { SNO SNO } }
```

它表示了 SP 的某个子集，即包含一些供应关系元组的子集，这些元组的 SNO 值与 *Sx* 的 SNO 相等。给出的表达式（即 SP MATCHING…）在逻辑上等价于映像关系引用“!! SP”。

5.4 a. 位于同一城市的供应商提供的零件号；

b. 没有提供零件 P2 的供应商；

c. 构成了 PNO–PNAME–WEIGHT–COLOR–CITY–SCT 元组，其中 PNO 的值由供应商 *n* 提供，这里 *n* 是 SCT 的一个值。

d. 所供应零件总重量小于 500 的零件号。

5.5 a. `( S WHERE CITY = MIN ( S { CITY } ) ) { SNO }`

b. `S { CITY } WHERE COUNT ( ‼S ) > 1`

5.6 单独的一个例子也应当给出整体的思路。考虑下面的表示式（本章中曾经提过）：

```
S WHERE COUNT (‼SP ) < 3
```

它可以等价于如下形式：

```
( ( EXTEND S : { X := COUNT (‼SP ) } ) WHERE X < 3 ) { ALL BUT X }
```

第6章

约束和断言

具有完整性和一致性——你的学分才能越积越多。

——摘自中国福饼中的信息（2004 年 11 月 15 日）

在数据库领域中，约束和断言是有区别的，但也是相关的，同时，它们又都是很重要的。约束用来保证数据库中的数据是正确的，断言用来处理数据的真正含义。下面我们仔细研究一下。

6.1 数据库约束

在第 3 章中我第一次提到数据库约束，虽然在那一章中把它们称为完整性约束。不严格地讲，数据库约束（或简称为约束）就是一个结果必须为TRUE的布尔表达式（因为它的结果为FALSE，说明数据库中肯定存在了错误的数据）。因此，任何修改数据库的请求如果使得约束判断结果为FALSE，就一定会失败。在关系模型中，立即失败意味着只要有修改请求，就会产生例外情况[1]。注意：这个要求有时被称为**指导原则**（约束破坏会立刻被检测）。可以立刻看到这个规则的作用结果，即没有一个用户曾经看到过数据库处于一种不一致的状态（不一致的状态是指至少有一种约束被破坏了的状态）。进一步说明一下，不一致状态肯定是一种不正确的状态，因此，它不可能与真实世界中的实际状态一致。我将在本章末尾来讲解这一点。

现在，我来讨论第 3 章介绍过的码和外码。这些约束是相当重要的，但也很简单。然而，也可能形成任意复杂的约束，而且在实际中也可能发生（有时把它们称为商业

1 这个要求确实是很明显的。我这里提到它只是因为有时确实允许修改数据库的一种状态，即使这个状态是不正确的（我们将会在本书第三部分看到解释），而且，甚至允许用户看到这个不正确状态。

规则）。首先，我用自然语言的形式给出几个约束的例子，然后再介绍这些约束在 **Tutorial D** 中如何表示。**注意：**下面的列表与第 3 章给出的例子相似，但不完全相同。

1. 供应商的状态值必须在 1 至 100 之间。
2. 伦敦的供应商状态值必须为 20。
3. 供应商号必须是唯一的。
4. 状态值小于 20 的供应商不能提供零件 P6。
5. SP 中的每个供应商都必须存在于 S 中。

现在我们考虑这些约束条件（或商业规则）如何采用 **Tutorial D** 的形式来表示。从第一个例子开始：

1. 供应商的状态值必须在 1 至 100 之间。

表达式为：

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

在 **Tutorial D** 中，约束条件通常采用 CONSTRAINT 语句来表示，就像例子中给出的，这个语句由三部分组成：（a）关键词 CONSTRAINT；（b）约束的名字（返回错误信息时或引用约束时使用）；（c）布尔表达式，必须取值为 TRUE。在这个例子中，布尔表达式的实际含义是如果我们把供应商关系的元组限制为状态值小于 0 或者大于 100，那么这个限制的结果就是空集（这个关系变量 S 的值是结果约束条件检测的）。

顺便提一下，如果检测约束时关系变量 S 恰好是空的，会发生什么呢？（答案：这个约束也是要满足的。）

实际上，还有另一种方法来表示前面的约束，这种方法虽有些争论，但从直觉上来说是更令人满意的（也许具有更好的用户友好性）。具体的表示形式如下：

```
CONSTRAINT CX1 AND ( S , STATUS ≥ 1 AND STATUS ≤ 100 ) ;
```

解释如下。

- 这里的 AND 是聚集运算符（我在第 5 章曾提到这个运算符，但没有给出例子）。这个聚集作用于布尔值。在这个例子中，聚集运算符包含一个布尔值，该值来自于供应商关系的每个元组，由布尔表达式 STATUS ≥ 1 AND STATUS ≤ 100 判断后得到。
- 供应商关系包含了 S1、S2、S3、S4 和 S5 的元组，相应的状态值分别为 20、10、30、20 和 30。因此，相应的布尔值分别为 TRUE、TRUE、TRUE、TRUE 和 TRUE。
- 现在，当且仅当相应聚集运算中所有值都为 TRUE 时，AND 返回值为 TRUE，就像此题给出的例子，每个供应商的状态值都在所给范围之内。

所以，这种替代的表示方式也是正确的[1]。我说这种表示方式比前一种更具有用户友好性的原因是它以正确的方式表明了要满足的条件（即是我们所希望的方式，这个状态值必须在一个特定的范围内）。相反，前面的那种表示方式是以一种相反的方式来说明要满足的条件，即这些状态值不能处于什么样的范围。

2. 伦敦供应商的状态值必须为 20。

表达式为：

```
CONSTRAINT CX2 IS_EMPTY ( S WHERE CITY = 'London' AND STATUS ≠ 20 ) ;
```

替代的表示方式为：

```
CONSTRAINT CX2 AND ( S , CITY ≠ 'London' OR STATUS = 20 ) ;
```

可以在两种表示方式中任选其一。

3. 供应商号必须是唯一的。

```
CONSTRAINT CX3 COUNT ( S ) = COUNT ( S { SNO } ) ;
```

这种表示方式没有明显表示出我们所要求的条件，但实际上它做到了。假设关系变量 S 的当前值为 *s*（当然也是一个关系），那么（a）表达式 COUNT (S)的就是 *s* 的度；（b）表达式 COUNT (S{SNO})就是 *s* 在 SNO 投影的度；（c）约束要求两个度要完全相等。（如果这种解释还不是很明显，即要求两个数相等其实等价于要求供应商号是唯一的，那就要根据关系变量 S 中的样本数据来进行解释了。）

实际上，我们几乎可以肯定的是这个例子中的约束条件其实不用采用 CONSTRAINT 语句来说明，相反，可以在定义相应关系变量时采用恰当的 KEY 语句说明（就像例子中给出的关系变量 S）。但有趣的是，这种表示方式太冗长了，因此，有时采用 CONSTRAINT 这种简化方式。

4. 状态值小于 20 的供应商不能供应零件 P6。

```
CONSTRAINT CX4
    IS_EMPTY ( ( S JOIN SP ) WHERE STATUS < 20 AND PNO = 'P6' ) ;
```

这个例子中涉及了两个不同的关系变量 S 和 SP。通常情况下，一个约束中可能涉及一些不同的或不相关的关系变量。术语：如果一个约束中只涉及一个关系变量，则称为单变量约束；如果约束中涉及两个或多个关系变量，则称为多变量约束。（因此，约束 CX1～CX3 都是单变量约束，而 CX4 是多变量约束。）

5. SP 的每个供应商都必须存在于 S 中。

```
CONSTRAINT CX5 SP { SNO } ⊆ S { SNO } ;
```

---

1 但是，当检测约束条件时，关系变量 S 恰好又为空，那么又会发生什么情况呢？

这个例子中也涉及了多变量约束。更重要的一点是，该例子中的约束条件也可以不使用 CONSTRAINT 语句表示，而是采用 FOREIGN KEY 的形式在定义关系变量时说明（例如该例中的 SP）。但是和例子 3 一样，采用 CONTRAINT 语句说明其实就是一种简化形式。

对于约束的例子说明就到此为止。但关于约束的讨论不仅仅如此，其实还有很多。然而，本书属于介绍性的一本书，不可能进行深入的讲解，所以我在本书中没有做进一步的解释说明。可以在SQL and Relational Theory一书中找到对这个主题更详细的讲解。同时，对于特定约束更加详细的检测又称为依赖（*dependencies*）即函数依赖或联接依赖），该部分内容可以在我的另一本书中找到，即Database Design and Relational Theory: Normal Forms and All That Jazz（2012 年，O'Reilly出版）[1]。

关于多重赋值的解释：这里我至少还要提到一点（虽然在本书中对此做过多解释不太合适），假设供应商-零件数据库具有如下约束，即供应商 S1 和零件 P1 不能位于同一个城市中（参看本章练习 6.3 的 e），那么，如果供应商 S1 和零件 P1 当前都位于伦敦，下面的 UPDATE 操作将会失败：

```
UPDATE S WHERE SNO = 'S1' : { CITY = 'Paris' } ;

UPDATE P WHERE PNO = 'P1' : { CITY = 'Paris' } ;
```

为了同时把 S1 和 P1 移到巴黎，可以说我们需要能够同时修改关系变量 S 和 P。为达到此目的，提供了一个多重赋值，它允许在同一个语句中同时对多个不同的变量进行修改，形式如下：

```
UPDATE S WHERE SNO = 'S1' : { CITY := 'Paris' } ,
UPDATE P WHERE PNO = 'P1' : { CITY := 'Paris' } ;
```

（注意这个逗号分隔符，它表明两个 UPDATE 是同一个语句中的一部分。）当然，UPDATE 是真正的赋值，所以前面的“两个 UPDATE”可以简写为如下的两个赋值语句：

```
S := ... , P := ... ;
```

这两个赋值语句把一个值赋给关系变量 S，另一个赋给关系变量 P，但它们是整个操作的一部分，直到整个语句结束才执行完整性检查（即直到分号才结束）。

现在来考虑我们刚才讨论的那个约束，即供应商 S1 和零件 P1 不能位于同一个城市，它实际上是一个多变量约束。我相信你也明白了，上面讨论的内容同样适用于很多这样的多变量约束，但不完全都适用。例如，他们肯定适用于约束条件 CX4。关于此问题的进一步讨论，请参见 SQL and Relational Theory 一书。

---

1 函数依赖（不是联接依赖）将在本书第 9 章简要介绍。

6.2 关系变量断言

现在我想改变一下话题，可以说是转向本章的另一个大的主题，即断言。这个主题的本质是采用另外一种方式来考虑变量。我的意思是说，大部分用户都只把关系变量看作是传统计算意义上的文件，相反，允许它是抽象文件（遵守规则的[disciplined]可能比抽象更合适），虽然如此，但它仍然是文件。只是我们以另外一种方式来看待它，通过该方式，我们可以对关系变量所起的作用有更深层次的了解，具体说明如下。

考虑供应商关系变量 S，像所有的关系变量一样，假设这个关系变量是真实世界中的某些值。实际上，更准确一点说，这个关系变量的标题表示了一种确定的断言，其含义是它是对真实世界中某些值的一种通用表示（说它是通用的，是因为它是参数化的）。刚才讨论的这个问题的断言如下：

Supplier SNO is under contract, is named SNAME, has status STATUS, and is located incity CITY.

这个断言就是对关系变量 S 的预期解释。换句话说，也可以称为内涵（intension）。

通常情况下，你可以把断言看作一个真值函数。像所有的函数一样，它有一系列的参数，当被调用时返回一个结果，该结果可以为 TRUE 或 FLASE。例如，上面的这个断言参数为 SNO、SNAME、STATUS 和 CITY（与相应关系变量的名称一致），它们代表了相应类型的数值（分别为 CHAR、CHAR、INTEGER 和 CHAR）。当调用这个函数时（按照逻辑学家的说法，是初始化断言），我们用变量值替代参数，假设替代值分别为 S1、Smith、20 和 London，那么就会获得如下的语句：

Supplier S1 is under contract, is named Smith, has status 20, and is located in city London.

这个语句实际上是一个命题（*proposition*），从逻辑上来说，它的值是很清楚的，要么为真，要么为假。下面给出几个例子：

1．Barbara Kingsolver 撰写了 *The Poisonwood Bible*。

2．Jane Austen 撰写了 *The Poisonwood Bible*。

第一个是真的，而第二个是假的。不要认为命题始终都是真的！然而，现在我们所讨论的命题都假设是真的。解释如下。

- 首先，显然每个关系变量都是与之相关的断言，称为关系变量断言（*relvar predicate*）。（该部分前面给出的断言就是关系变量 S 的关系变量断言。）
- 关系变量 *R* 具有断言 *P*，那么在某个时刻出现在 *R* 中的每个元组 *t* 都可以

被看作是表示了一个特定的断言 *P*，该断言是把元组 *t* 中的属性值作为变量值调用得到的。

- （这一点非常重要。）按照惯例，我们假设采用这种方式获得的断言 *P* 判断结果都为 TRUE。

还是以关系变量中的样本数据为例，此时，假设下面的所有断言都为 TRUE：

Supplier S1 is under contract, is named Smith, has status 20, and is located in city London.

Supplier S2 is under contract, is named Jones, has status 10, and is located in city Paris.

Supplier S3 is under contract, is named Blake, has status 30, and is located in city Paris.

而且，我们进一步讨论，如果在某个给定的时间 *t*，某个特定的元组貌似有理由应该出现在某个特定的关系变量，但实际上没有出现，那么我们在时间 *t* 的那个断言就是错误的。例如下面的元组：

```
TUPLE { SNO 'S6' , SNAME 'Lopez' , STATUS 30 , CITY 'Madrid' }
```

它貌似有理由是一个供应商元组，但是在时间 *t* 没有出现在关系变量 S 中，所以我们有权假设下面的这个断言在当时就是错误的：

Supplier S6 is under contract, is named Lopez, has status 30, and is located in city Madrid.

总结一下，一个给定的关系变量 *R* 在任意给定的时间包含所有的、而且只包含那些值为真的断言或者至少我们在实际中假设为真的断言。换句话说，我们实际上采用的是封闭的假设区域。下面给出定义：

定义：关系变量 *R* 具有断言 *P*，那么封闭的假设区域（**The Closed World Assumption**，CWA）指的是（a）如果元组 *t* 在时间 *T* 出现在 *R* 中，那么假设与 *t* 对应的 *P* 的初始值 *p* 在时间 *T* 是正确的。反过来，（b）如果元组 *t* 与 *R* 的标题一致，但在时间 *T* 没有出现在 *R* 中，那么假设与 *t* 对应的 *P* 的初始值 *p* 在时间 *T* 是错误的。换句话说，元组 *t* 在给定的时间出现在关系变量 *R* 中，当且仅当它在那个时间满足 *R* 的断言。

更多的术语：*P*代表关系变量*R*的断言，在某个给定的时间*R*的值为关系*r*，那么*r*（更确切地说，是*r*中的内容）就是那个时间*P*的扩展[1]，因此，对于给定的关系变量，它的扩展是随着时间变化的，但是内涵却不变。

---

1 它与第 5 章讨论的 EXTEND 运算符无关。

关系与类型

综上所述，在任何给定的时间，数据库都可以看作是正确断言的集合，例如，断言 *Supplier S1 is under contract, is named Smith, has status 20, and is located in city London*，出现在这个断言中的变量（如 S1、Smith、20、London）都恰好是与其对应的元组的属性值，每个属性值都是具有特定类型的值。它遵循如下规则：

> 类型是我们讨论的相关事情的集合；关系就是描述我们要讨论的事情的值为真的语句。

换句话说，类型给我们提供了词汇表，关系赋予了我们讨论这些事情的能力。例如，为了简化，假设我们只把注意力放在供应商关系上，那么：

- 我们讨论的事情可以是字符串、整数或其他（当然，在实际的数据库中，我们的词汇表通常比这个要大，特别是把用户定义的类型包括进来的时候）；
- 我们可以说的东西形如 “用特定字符串表示的供应商号必须符合约束的规定，供应商名字采用字符串表示，状态值采用整型值表示，所在的城市采用字符串表示”，或者其他。

这个事件的当前状态至少有三个重要的推论。具体地说，是为了表示真实世界的某个部分。

1．类型和关系都是必需的。没有类型，我们就不能讨论事情；没有关系，我们就不能说具体的东西。

2．类型和关系就足够了，也是必须的。从逻辑上来说，我们不再需要其他的了[1]。（为了反映真实世界中的多种变化，我们确实需要关系变量，但是我们不需要用它去表示任意给定时间的状态。）

3．类型和关系是不一样的。当心那些把它们混为一谈的人！事实上，把类型看作是关系的某种类型就相当于让非关系产品去执行一些关系操作（虽然不用特殊说明，但它们通常不采用这样的术语来讨论）。很明显，建立在这种逻辑错误上的任何产品注定最终都将会失败。

下面再从比较规范的角度来解释一下上面所讨论的事情。正如我们所看到的，数据库可以被看作是值为真的断言的集合。事实上，数据库与运算符合在一起就是一个逻辑系统，这些运算符被赋予表示数据库的那个断言。“逻辑数据库”的含义

---

1 当我说类型和关系是必需的、足够的时候，我当然要提到第 1 章提到的逻辑数据库。显然物理数据库中需要其他的结构（如指针），这是因为它们的设计目标不同，处于不同的级别上。物理模型显然已经超出了关系模型的范围。

就是一个规范的系统（如欧几里得几何），我们可以利用公理（给定的事实）或推理规则来证明一些定理（推导出的事实）。这些都来自于Codd伟大的洞察力，他在1969 年发明了关系模型，证明了数据库并不真正是数据的集合（虽然名字叫做数据库），相反，它是一些事实的集合，或者称为断言。这些断言（也就是说，给定的断言就是在基本的关系变量中由元组表示的那些 [1]）就是逻辑系统中的公理。推导规则就是基本的规则，利用这些规则可以从已有断言中推导出新的断言。换句话说，这些规则告诉我们如何使用关系代数中的运算符，因此，当这个系统要判断某些关系表达式时（特别是要对某些查询做出响应），就要产生一些新的规则。实际上，这是在证明定理。

一旦你理解了前面所讲的，你就会发现这种规范逻辑表示可以用来解决“数据库问题”。换句话说，形如下面的问题在逻辑操作上都是可以接受的，都可以给出逻辑答案。

- 从用户的角度看，数据库应该是什么样子的？
- 完整性约束应该是什么样子的？
- 查询语言应该是什么样子的？
- 我们应该怎样最好地实现查询？
- 通常情况下，我们如何更好地评价数据库表达式？
- 运算结果应该如何呈现给用户？
- 我们首先应该如何设计数据库？

现在，我们没有说关系模型可以直接支持前面的概念，这是因为模型是坚如磐石的、正确的，可以持续的。另外也因为其他所谓的“模型数据”不只是简单的在同一个范围内。实际上，我一直有个疑问，即那些其他的模型是否能被称为数据模型，至少不能在关系模型的意义上这样来命名。可以肯定的是，在集合理论和断言的逻辑上，它们中的绝大部分都是可以作为关系模型的 [2]。

6.3　断言与约束

断言和约束两个概念之间有联系吗？我们再次考虑一下关系变量 S 的断言：

Supplier SNO is under contract, is named SNAME, has status STATUS, and is

---

1　回顾第 3 章，基本关系变量就是可以单独存在的关系变量，不能由其他关系变量来定义。像 S、P 和 SP 都是基本关系变量。

2　虽然集合理论的相关性很明显，但这是我第一次在本书中提到集合理论和断言逻辑（参见附录 C），至于断言逻辑，可以参见附录 D。

located in

city CITY.

这个断言是专门为关系变量 S 解释的。在理想的世界里，它可以作为修改关系变量的一个可以接受的标准。也就是说，它可以决定在关系变量 S 上进行的修改是否可以被接受。但是这个目标是不能实现的。

- 一个原因是，系统不能识别“一个供应商”是否“遵守协议”或者“位于哪里”的具体含义，即这是由解释来负责的。例如，如果供应商号 S1 和城市名 London 恰好同时出现在同一个元组中，那么用户就可以做出解释，即供应商 S1 位于伦敦，或者供应商 S1 经常位于伦敦，或者供应商 S1 在伦敦有办公室，再或者供应商 S1 在伦敦有房产，或者任意其他可能的解释（当然，要与可能的关系变量断言的无限的数量相对应），但是系统是没有办法完成类似事情的。
- 另一个原因是，即使系统知道供应商应遵守协议或位于哪里等的具体含义，但它仍然不知道这个前提条件，即用户所说的是否是真的！如果用户通过修改操作保持系统的一致性，例如，供应商 S6 名字叫做 Lopez，状态值为 30，所在城市为 Madrid，那么系统也没有办法知道这个断言是否是真的。系统能够做的所有的事情是检测用户的断言不能引起任何完整性约束的破坏。假设不是这样的话，系统就会接受用户的断言，会从那一时刻开始把它看作是真的（直到这时用户才会通过执行另一个修改操作来告诉系统，这个断言不再是真的）。

因而，与理想的情况相反，修改可以被接受的实际标准不再是断言，而是与之相应的约束集合，这些约束可以被看作是相应断言的一种近似表示。等价的说法为：

系统不能加强真理，而是要加强一致性。

令人遗憾的是，真理和一致性不是一回事。具体地说，如果数据库会包含真的断言，那么它就是一致的，但反过来不一定是这样。相似地，如果它不一致，那么它至少包含一个错误的断言，但反过来不一定是这样。或者换一种说法，正确就暗含着一致性，不一致就暗含着不正确，那么：

- 数据库是正确的，就是指它能够真实地反映出真实世界中的事件的正确状态；
- 相反，数据库是一致的，只是说它没有破坏任何已知的完整性约束。

6.4 练习

6.1 针对关系变量 P 和 SP，给出一些合理的断言。

6.2 考虑本章给出的约束 CX1 至 CX5：

CX1：供应商的状态值必须在 1 至 100 之间。

CX2：伦敦供应商的状态值必须为 20。

CX3：供应商号必须是唯一的。

CX4：状态值小于 20 的供应商不能供应零件 P6。

CX5：SP 的每个供应商都必须存在于 S 中。

哪些操作有可能会破坏这些约束？注意：不必用非常准确的形式给出你的答案，例如，针对约束 CX1 的操作，“在 S 中进行插入”作为答案这就足够了。

6.3 针对下面的商业规则，写出供应商-关系数据库的 CONSTRAINT 语句：

a．所有红色零件的重量必须大于 50 磅。

b．不能出现两个供应商位于同一个城市的情况。

c．在任何时候，最多只能有一个供应商位于雅典。

d．至少存在一个伦敦的供应商。

e．供应商 S1 和零件 P1 必须位于同一个城市。

6.4 在关系变量 S 当前值上，对除了 CITY 外的所有属性进行投影，这个投影包含了所有形如（*sno,sn,st*）的元组，这样的元组来自于关系变量 S 的元组（*sno,sn,st,sc*）（我已经尽量简化地来解释这些概念）。换句话说，讨论的这个投影有一个与如下形式对应的断言：

Supplier SNO is under contract, is named SNAME, has status STATUS, and is located somewhere

供应商可能位于某个城市，但我们不知道具体的城市。

注意，这个断言有说哪个参数与投影中的三个属性相对应。

类似的结论可应用于所有可能的关系表达式中。换句话说，它不只是具有断言的关系变量，相反，每个关系表达式都有一个断言[1]。而且，对于给定关系表达式的断言可以一直由表达式中涉及的关系变量的断言决定，其语义与表达式中关系运算符的语义一致。请给出下面表达式的断言：

a. `( S JOIN P ) { PNO , SNO }`

b. `P MATCHING S`

c. `S NOT MATCHING ( SP WHERE PNO = 'P2' )`

d. `EXTEND P : { SCT := COUNT ( ‼SP ) }`

e. `P WHERE SUM ( ‼SP , QTY ) < 500`

---

1 实际上，用户可能还没有意识到这个事实，他们写出一个规范的查询之后，就要解释这个查询的结果（可以采用 Tutorial D 或其他语言书写查询）。

6.5 在本章开头，我曾经声明“黄金法则”是显而易见的，但真的是显而易见的吗？如果它被破坏的话，可能会出现什么状况？

6.5 答案

6.1 关系变量P的断言：*Part PNO is named PNAME, has color COLOR and weight WEIGHT, and is stored in city CITY*[1].

关系变量 SP 的断言：*Supplier SNO supplies part PNO in quantity QTY*.

6.2 答案如下：

CX1：INSERT on S, UPDATE on STATUS in S

CX2：INSERT on S, UPDATE on STATUS or CITY in S

CX3：INSERT on S, UPDATE on SNO in S

CX4：INSERT on SP, UPDATE on SNO or PNO in SP, INSERT on S, UPDATE on STATUS in S

CX5：INSERT on SP, UPDATE on SNO in SP, DELETE on S, UPDATE on SNO in S

6.3 答案如下：

a. `CONSTRAINT CXA AND ( P , COLOR = 'Red' AND WEIGHT < 50.0 ) ;`

b. `CONSTRAINT CXB COUNT ( S { SNO } ) = COUNT ( S { CITY } ) ;`

c. `CONSTRAINT CXC COUNT ( S WHERE CITY = 'Athens' ) < 2 ;`

d. `CONSTRAINT CXD IS_NOT_EMPTY ( S WHERE CITY = 'London' );`

e.
```
CONSTRAINT CXE COUNT ( ( S WHERE SNO = 'S1' ) { CITY }
                       UNION
                       ( P WHERE PNO = 'P1' ) { CITY } ) < 2 ;
```

6.4 答案如下：

a．供应商 SNO 和零件 PNO 属于同一个城市；

b．零件 PNO 的名称为 PNAME，颜色为 COLOR，重量为 WEIGHT，位于城市 CITY，并且只能由一个供应商来供应。

c．供应商 SNO 遵守一定协议，名字为 SNAME，状态值为 STATUS，位于城市 CITY，但不提供零件 P2。

d．零件 PNO 的名称为 PNAME，颜色为 COLOR，重量为 WEIGHT，位于城市 CITY，由供应商 SCT 来供应。

e．零件 PNO 的名称为 PNAME，颜色为 COLOR，重量为 WEIGHT，位于城

---

1 也许更确切地说，零件 PNO 是用在企业中的，无论是什么企业均可， PNAME 也是如此。

市 CITY，由供应总量大于 500 的供应商来提供。

注意：这个练习只是对前面章节的练习更规范的一种表示（如第 4 章的练习 4.6）

6.5 真理即为：我们从不相信从不一致的数据库中取得的答案。下面给出这个事实的证明过程（该部分的讨论主要基于 *SQL and Relational Theory* 一书）。我们知道，数据库可以被看作是断言的集合。假设这个集合是不一致的，那么假设采用明显的或暗示的方式说明了一些断言 *p* 和它的反向结果 *NOT p* 都是正确的，*q* 表示任意的断言，那么：

- 从真理 *p* 中我们可以推断出真理 *p* 或 *q*。
- 从真理 *p* 或 *q*，或 *NOT p*，我们可以推测出真理 *q*。

但 *q* 是任意的！即无论什么情况下，在一个不一致的系统中任意断言都可以视为是“真”的（即使这个断言很明显是错误的，如 1=0）。

第 7 章

关系模型

关系模型：规范的信息！

——Anon：*Where Bugs Go*

这是本书第一部分的最后一章。在前面的章节中，我已经描述了关系型 DBMS 的大部分特征（至少从关系型的视角来看，这些特征是比较重要的）。现在我们复习一下这些特征，并把它们综合在一起来定义关系模型，以此来解释这些特征的具体含义。有一点要提醒的是，术语关系模型（relational model）本身来自于 Codd 的一篇著名的论文题目 *A Relational Model of Data for Large Shared Data Banks*（CACM13，第 6 期，1970 年 6 月），此篇论文在附录 A 中有详细介绍。

7.1 关系模型定义

我不知道你是否注意到了，在该章之前我故意没有给出关系模型的确切定义[1]，尽管我在许多场合似乎提到了这个概念，比如：

- 关系模型没有元组级的运算符；
- 关系模型禁止使用指针；
- 关系模型需要立即进行完整性检测。

我真正给出的最相近的定义是在第 1 章，把关系模型描述成一个用户接口，这是事实，但它也不能作为一个确切的定义。现在，我把关系模型看作是 DBMS 的一个抽象的外部说明，但在有些华而不实的语言中仍然采用“接口”的定义。我们尤其注意到，这两个定义都没有明确说明给定的 DBMS 是否是关系型的。

---

1 有趣的是，尽管 Codd 在 1970 年发表的论文标题给出了关系模型，但实际上并没有真正给出关系模型的定义。

实际上，从直觉上来说，我把它看作是一种可以简单而简洁解释说明的分量最重的一种模型（即使已经定义了）。为了支持这个论点，考虑下面这个经过简单编辑后的文本，这个是我写的一个网页，描述了 Codd 的背景及贡献，并对他在 1981 年获得的 ACM 图灵奖给出了解释（*http://amturing.acm.org/award_winners/codd_1000892.cfm*）。

数据的关系模型

实际上，数据的关系模型是对数据库中数据的一种规范而严格的定义，即数据呈现给用户时是什么样子的。例如，它是用户与数据库系统之间接口的一种抽象的说明。换句话说，数据库系统是关系型的，当且仅当它支持的用户接口真正实现了关系模型。就这样的用户系统而言，应该满足:（a）数据看上去是关系型的;（b）像联接（join）这样的关系运算符应该可以作用于关系型的数据。

数据看上去是关系型的

关系在纸上可以表示为简单的具有行和列的表格。因此，不严格地说，用户可以采用这样的表格形式来看待数据。

可用的关系运算符

关系运算符就是可以从给定的关系中再产生新的关系的运算符。因此关系可以被看成是表格，所以我们可以把这些运算符看作是对表格进行的“剪切和粘贴”操作。例如，给定一个员工信息表格，限制（restrict）运算就是根据给定的工资裁剪掉某些员工的行。如果再给出一个部门信息表格，联接（join）运算符就是把员工的部门信息与员工信息粘贴在一行中。

实现

因为关系型系统的用于接口与系统内部数据的存储和操作方式完全不同（要比系统内部的方式简单一些），所以在实现时会面临很多挑战。早在 20 世纪 70 年代研究人员就已经开始研究这个问题了，不久，Codd 发表了他的第一篇论文，许多的原型都是在 20 世纪 70 年代提出的。20 年后出现了第一个商业化产品。然而不幸的是，即使到现在，也没有一个商业化产品能够真正实现 Codd 的最初想法，但它们很快就主宰了市场。由于关系模型是建立在稳固的集合理论和规范逻辑基础上，至今它仍然存在，而且能接受时间的考验。

虽然关系模型被定义得很简单，但它还是很深奥的。别忘了，我已经用了一百多页的篇幅来描述它的深奥，我的做法肯定不是很彻底的。因为即使过去了这么多年，我仍然发现了一些结果和新的含义（当我在 1970 年阅读 Codd 的论文时，第一次听说了关系模型）。简言之，我仍然是在学习过程中。

因此，在本章我想要给出关系模型的更确切的定义。但麻烦的是，我给出的定义真的是很准确的吗？所以，事实上，我认为要理解第 1 章给出的定义是非常困难的。（这就是我为什么没有做任何事情的原因。就像 Bertrand Russell 说过的值得纪念的一句话“作品或者通俗易懂、或者严格缜密，但不能同时做到”。）所以，我给出定义后，要必须花费大量篇幅来加以解释。

不再多费周折，下面是我给出的定义：

定义：关系模型由以下 5 个部分组成[1]：

1．类型的集合，该集合是开放的，并且是闭集。尤其包括 BOOLEAN 类型。
2．关系类型产生器，以及对产生类型的详细解释。
3．定义关系变量的工具，这些关系变量的类型是由关系类型产生器确定的。
4．关系赋值运算符，可以把关系值赋给确定的关系变量。
5．从关系上说，是完整的，但是开放的通用关系运算符的闭集，这些运算符可以从其他关系值产生新的关系值。

下面按次序来解释这几个部分。但首先要注意一点，John Muir 曾经说过，当我们试图从事物本身找出什么的时候，就会发现它与宇宙中的一切都有联系（经常以这种形式引用“任何事物都与其他事物有联系”）。当然 John Muir 是在讨论自然界，但他也一直在谈论关系模型，即关系模型的各个特征之间是高度关联的，删除其中任何一个，整个结构就会被破坏。采用术语解释的话，这个隐喻的含义就是如果我们建立的“关系型”系统不能支持该模型的某个方面，那么这个系统（也许不能把它称为真正的关系型系统）就不能解释某些场合的一些行为，这肯定是我们不希望发生的，但也不是不能预见的。对于该点我不能强调太多，模型中的每个特征都有它存在的实际原因，如果我们忽略任何一个，就是在给自己找麻烦。

7.2 类型

再重复一遍，5 个部分中的第一个特征是“类型的集合，该集合是开放的，并且是闭集。尤其包括 BOOLEAN 类型”。关系模型中肯定是需要类型的，因为关系是定义在类型的基础上的。每个关系的每个属性都属于某种类型，因此，每个关系变量也应该属于某种类型。

但也要特别注意，关系模型并没有说存在什么类型（只是特别提出了 BOOLEAN）。

1 顺便说一下，一定要注意这个明确的冠词 *the* relational model，即特指关系模型，仅仅有这一种表示！一些批评家有时会建议说，关系型的提倡者（这其中要包括我自己）也在改变这个定义，改变规则。为此，2006 年我发表了论文 *There's Only One Relational Model*，对这种批评做出了回应。

通常会有一种误解，认为关系模型只能处理一些相当简单的类型，如数值型、字符串、日期型、时间型等等，这是很不幸的。但实际上关系和关系变量都具有属性，这些属性可以具有任何类型（除了下面列出的）。换句话说，支持类型与支持关系模型本身是正交的。

现在，开始讨论类型。这里讨论更多的是标量类型（scalar type）（实际上，在我其他的著作中，我经常用术语标量类型［scalar type］来代替类型［type］）。那么到底什么是标量类型呢？不严格地讲，如果没有用户可见的部分，它就是标量的，否则就是非标量的。根据类型 T 本身是否是标量的来确定该类型 T 的值、变量、属性、运算符、参数和表达式。例如：

- 类型 INTEGER 是标量的（它没有用户可见的部分）。因此，INTEGER 的值、变量等等也都是标量的。
- 关系类型是非标量的（具有与相应属性一致的用户可见部分），因此该关系的值、变量等等也都是非标量的。注意，元组类型、值和变量也是非标量的（回顾第 2 章练习 2.7 的答案）。

现在必须强调一点：前面的概念实际上是不规范的、也不精确的，然而对于直观的理解是有帮助的。一部分原因是有一些类型实际上很难确定是标量还是非标量。（例如，字符串或者单独的字符都是有用户可见的部分组成的。）针对这样的原因，无论在何处，关系模型都信赖采用规范形式表示的标量与非标量类型之间的区别[1]。为了达到前面的目的，你可以采用术语标量（scalar）来表示除元组合关系类型之外的任何类型，也可以采用术语非标量（nonscalar）代表元组或关系类型。除此之外，你可以采用术语类型来特别表示标量类型，除非有显式的声明。

现在，标量类型可以是一个系统，也可以是用户定义的。因而，对于用户来说，可以定义自己的标量类型。用户也可以定义自己的运算符，因为没有运算符的类型是不能使用的。至于系统已定义的类型，这个类型集合中需要包括 BOOLEAN（它是最基本的类型之一，只有 2 个值，即 TRUE 和 FALSE），但一个实际的系统中还要支持其他的类型，如 INTEGER、CHAR 等。支持 BOOLEAN 的含义就是支持通用的连接词（如逻辑运算符 AND、OR、NOT 等）以及其他的、由系统或用户定义的、返回布尔值的运算符。特别是，等价比较运算符“=”（它也是一个布尔运算符）可用于与任何类型的联接运算中，它既是非标量类型也是标量类型，因为没有它，我们甚至不能说构成这些类型的值都是什么。而且，模型还规定了这个运算符的语

---

1 这就是我为什么没有在前面章节提到这个区别的原因。实际上我只是两次简单地提到了术语标量（scalar），一次是在第 3 章讨论标量标识符，一次是在第 4 章提到标量运算符（但只是在脚注中），而且我从没提到过非标量（nonscalar）。

义：如果 *v1* 和 *v2* 具有相同的值（它们的类型必须相同），那么等式 *v1=v2* 就返回 TRUE，否则返回 FALSE。

现在我们已经得出建议，关系和关系变量可以具有任何类型的属性不是 100% 正确的。事实上（就像第 2 章练习 2.7 的答案），会有两种例外，首先，在数据库中关系变量和关系不允许具有指针类型；第二，如果关系 *r* 具有标题 *H*，那么就不能依据具有同样标题 *H* 的元组或关系类型来定义 *r* 的属性。

最后，要注意的是与每个类型 T（无论是标量还是非标量）有联系的是必须至少存在一个选择器运算符（说明这一点主要是为了完整性，第一次阅读时如果不理解，也不重要）。选择器是只读运算符，具有如下属性：（a）每次调用该运算符都返回类型为 *T* 的值；（b）类型 T 的每个值都是通过调用该运算符来返回（更确切地说，是通过相应的标识符。注意，标识符是选择器调用的一种特殊情况）。因此，给定任意类型 *T*，必须能写出一个标识符来表示类型 *T* 的任意值。详细解释请参照 *SQL and Relational Theory*。

7.3 关系类型产生器

模型的第 2 部分是关系类型产生器，以及对产生类型的详细解释。这是在本书中第一个词提到术语类型产生器（type generator），所以我先岔开话题解释一下这个术语。类型产生器基本上只是一种特殊类型的运算符，它的特殊是因为（a）它返回一个类型，而不是一个值；（b）它在编译时调用，而不是在运行时调用。例如第 1 章图 1.3 所示的代码段包含了下列 VAR 语句：

```
VAR A ARRAY [1..N] OF INTEGER ;
```

这个语句定义了一个变量 A，它的值是一个一维数组（通过 A[1], A[2], …, A[N] 的形式引用），数组中每个元素都是整数。那么表达式 ARRAY [1..N] OF INTEGER 就被看作是对 ARRAY 类型产生器的调用，它返回一个特定的数组类型。这个特定的数组类型就是一个被产生的类型（它产生的是一个非标量类型，虽然有时也可能产生标量类型）。因此，请注意在类型产生器和类型本身之间是有逻辑差异的，更确切地说，类型产生器不是一个类型。

再回到关系模型，这里我们最感兴趣的类型产生器当然是 RELATION 类型产生器，这个类型产生器允许用户说明单个的、所希望的关系类型。下面举几个例子（采用 **Tutorial D** syntax 语法）：

```
RELATION { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }

RELATION { PNO CHAR , PNAME CHAR , COLOR CHAR ,
```

```
                                          WEIGHT RATIONAL , CITY CHAR }

RELATION { SNO CHAR , PNO CHAR , QTY INTEGER }
```

当然，这些类型分别是关系变量 S、P 和 SP 的类型，也是我们正在使用的例子。换句话说，RELATION 类型产生器的一个重要作用就是在定义一些关系变量时（如在 **Tutorial D** 中的 VAR 语句）去说明所定义关系变量的类型（事实上，也是最重要的用途）。

关系值属性：这一点已经超出了本书的范围，我只是想作为过渡来提一下。我们知道：（a）属性是有类型的；（b）关系类型也是类型。所以了解一下关系能够具有来自关系类型的属性也是不足为奇的（例如，关系值属性，即属性的值也是关系 [1]）。因而，RELATION 类型产生器的另一个作用是说明属性的类型，这些属性的标题与关系标题一致。详细讨论参见*SQL and Relational Theory*。

至于商业中特定关系的解释，也是在给定的上下文环境中，给定类型的特定关系组成一组命题的集合。这样的命题满足：（a）由一组实例化的断言组成，这组断言与给定关系的标题一致；（b）由关系中的一个元组来表示；（c）这个命题假设为真的。如果讨论的上下文环境为一些关系变量（即如果我们讨论的关系恰好是这些关系变量的当前值），那么讨论的这个断言就是那个关系变量的断言。

而且，关系变量和关系的解释要与*The Closed World Assumption* 或CWA一致（参见第 6 章）[2]。

最后，让 *RT* 代表某种关系类型。那么，与 *RT* 有关的，有一个具有某些属性的选择器运算符（当然，这里尤指关系选择器（relation selector），即（a）每次调用该运算符都返回一个类型为 *RT* 的关系；（b）类型 *RT* 的每个关系都是在调用该运算符时返回的（更明确地说，通过调用关系标识符，参见第 3 章）。下面给出几组例子：

```
RELATION { TUPLE { SNO 'S1' , PNO 'P1' , QTY 300 } ,
           TUPLE { SNO 'S5' , PNO 'P6' , QTY 350 } }

RELATION { TUPLE { SNO SX , PNO PX , QTY QX } }
```

每个表达式都返回一个类型为 RELATION {SNO CHAR, PNO CHAR, QTY INTEGER}的关系，这个值可以赋给关系变量 SP。第一个表达式表示了 2 个元组的关系，实际上是一个关系标识符。第二个表达式只是表示了一个元组的关系，不是

---

1 事实上，我间接地提到过这个可能性。当我在前面部分中提到，如果关系 *r* 具有标题 *H*，那么 *r* 中的属性就不能依据与 *H* 同样的关系类型来直接或间接定义。

2 第 6 章定义的 CWA 是对关系变量的引用，不是对关系的引用，但是我希望的是它能自然地扩充到关系（例如，查询结果）。

标识符。实际上，选择器调用就是一个标识符，当且仅当它所有的变量表达式都是标识符时。

最后，因为等价比较运算符“=”可以用在与任何类型的联接运算中，尤其是常用在与关系类型的联接运算中，所以关系包含运算符“⊆”也可以使用。如果关系 *r1* 和 *r2* 具有同种类型，那么 *r1* 包含在 *r2* 中，当且仅当 *r1* 是 *r2* 的子集时。

7.4 关系变量

模型的第 3 个部分是“定义关系变量的工具，这些关系变量的类型是由关系类型产生器确定的”。从逻辑上来说，它遵守前面的规则。也就是说，RELATION 类型产生器的主要作用是（就像前面那个部分说明的）在定义关系变量时用来说明关系变量的类型。关系变量只允许在关系数据库中使用，其他类型的变量，如标量变量或元组变量都不允许使用。（相反，在访问数据库的程序中，它们是可以使用的）。

数据库中只包含关系变量的说法是由 Codd 最早在 *The Information Principle* 一书中正式提到的。然而，我认为他自己也没有规范使用。相反，他经常采用如下方式来说明这个原理：

> 在任何给定的时间，数据库中的整个信息内容只能采用唯一的方式来表示，即在关系中元组的属性中来说明具体的数值。

这个原理是我在第 1 章对关系系统进行简化说明的一个重要依据。实际上，我听说Codd曾经在很多场合把它作为关系模型的一条基本原理。因此对该原理的任何破坏都会造成严重的后果[1]。为什么这个原理如此重要呢？答案与我在第 6 章给出的结论有密切关系，即在逻辑层面上，无论表示什么样的数据，关系都是必须的，而且具有表示数据的能力。换句话说，关系模型为我们提供了一切我们需要的事情，它也没有提供我们不需要的任何事情。

对此问题我想多解释几句。首先，如果有 *n* 种不同的表示数据的方法，那么我们就需要 *n* 个不同的运算符集合。（第 2 章时我曾提到过这一点。）例如，如果既有数组又有关系，那么就需要有数组运算符以及关系运算符。我们也要选择数据采用关系还是数组来表示，而在进行选择时，没有相关的指南可以帮助我们。所以，如果 *n* 大于 1，我们就需要实现多个运算符，并对它们进行归档、学习、了解、记住及使用。但是这些额外的运算符增加了复杂度，而不是加强了力量。如果 *n* 大于 1 时它没有什么可做的，如果 *n* 等于 1 时也没有什么不能做的（当然在关系模型中，

---

1 这里没有说明对象数据库、XML 数据库及更通用的非关系数据库都会破坏它。不幸的是，SQL 数据库也是如此（参见本书的第三部分），除非遵循一个合适的规则。

n 肯定等于 1）。而且，关系模型不仅为我们提供了唯一的表示数据的方法，而且这种方法相当简单。

对数据库变量的解释：就像 Hugh Darwen 和我在我们的著作 *Databases, Types, and the Relational Model: The Third Manifesto*（**2007 年，第三版**）中证明的：数据库不只是关系变量的容器，即使我们通常这样讨论它，相反，它是一个独立存在的变量！别忘了，它是可以被修改的，因此，根据定义它是一个变量。换句话说，从逻辑上来讲，作为一个整体，数据库本身就是一个变量，称作数据库变量（记为 *dbvar*）。换个角度，数据库关系变量真的是一种假象（即使这个假象在实际情况中相当有用，就像 Einstein 曾经说过的（虽然在不同的环境中），唯一的真变量就是数据库本身，在它的全部。这个概念在我的另一本书中有深度解释，即 *View Updating and Relational Theory: Solving the View Update Problem*（2013 年，O'Reilly 出版）。

顺便说一下，我不知道你是否注意到了，到目前为止我还没有确切地给出术语数据库的明确定义。而且，现在也不会给出（这个问题比它看起来要复杂得多）。然而，为了达到实用的目的，我上面的这些做法也许是最简单的，即数据库就是关系变量的容器（从关系模型的角度来看）。要提醒的是，如果给定每个关系变量就是断言，也给定这些关系变量的每个元组，而这些关系变量与相应断言的真实例化值相一致，那么就可以在任何时候把数据库看作是表示这些真命题的集合（就像第 6 章看到的）。

7.5　关系赋值

关系模型的第 4 部分是“关系赋值运算符，可以把关系值赋给确定的关系变量”。它要遵守前面第 2 条和第 3 条规则。这一点很简单：赋值运算符“：=”与等价比较运算符“=”相似，它可以应用在任何类型中（因为没有它，我们就没有办法把值赋给特定类型的变量），当然关系类型也不例外。运算符 INSERT、DELETE 及 UPDATE（也包括 D_INSERT 和 I_DELETE）也允许使用，而且是非常有用的，但严格来说它们只是一种简化，而且，如果支持关系赋值，（a）必须要特别支持多重关系赋值（参见第 6 章 6.1 节“数据库约束”部分后面的注释），（b）必须禁止使用赋值规则（*The Assignment Principle*）和黄金规则（**The Golden Rule**），具体如下：

- 赋值规则（参见第 1 章）说明，把值 *v* 赋值变量 *V* 后，比较 *v* = *V* 就可以获得 TRUE；
- 黄金规则（参见第 6 章）说明，所有的完整性约束必须满足语句边界值。

7.6　关系运算符

关系模型的第 5 部分是“从关系上说，是完整的，但是开放的通用关系运算符

的闭集，这些运算符可以从其他关系值产生新的关系值”。当然，运算符的集合就是关系代数，或者与代数逻辑上等价的事物。就像在第 4 章讲到的，运算符是通用的，实际上从某种意义上来说它可以应用到任何可能的关系中（例如，我们不需要一个联接运算符去联接供应商和供应关系，再定义一个联接运算符去联接部门和雇员关系）。确切地说，并没有明确说明必须支持哪一个运算符，但至少需要它们才能获得关系完整性这一特征。

关系完整性是衡量某种语言表达能力的基本标准。不严格地说，如果某种语言从关系上来说是完整的，则（a）可以用此种语言表示“所有可能的查询”；（b）而且，任何查询都可以正规地表示为一个表达式。因此，任何复杂的查询如果没有采用分支或循环的话也可以规范实现 [1]，更确切一点地说，如果某种语言L是完整的，那么采用L表示的表达式就允许依据关系代数表达式来定义关系。因而语言L至少要与关系代数一样强大。因此基本的关系完整性就是允许用户直接访问数据库（至少从理论上来说是这样的，虽然实际上可能并不是这样），而不用通过IT部门的其他途径。

> **关系运算的解释：**实际上，第一次给出关系完整性的定义是在 Codd 的一篇论文中（参见附录 E），Codd 说，一个语言是关系上完整的，当且仅当它至少与关系运算一样强大，而不是关系代数。关系运算是关系代数的一种替代品。它是基于谓词逻辑的另一种规范表示，而不是基于集合理论的，因此（a）它可以用来表示规范的查询、约束等等；（b）它的表示方式与代数是等价的（表示方式等价的意思是，对于代数的每一种表示方式，在逻辑表示上与运算是一致的，对于运算的每一种表示方式，在逻辑上与代数也是等价的）。
>
> 对于关系运算这里不想讲的太详细。选择代数和运算只是个人喜好的问题，以上讲的这些就足够了。有些人喜欢代数，而有些人喜欢运算。（有些问题采用关系代数表示更容易理解，而有些问题采用关系运算会更容易理解，二者之间没有好与不好之分。再重复一下，它们是等价的。）如果你感兴趣，想了解更多的内容，可以参见附录 D。

现在，对于代数的目的似乎造成了更大范围的误解。特别是许多人似乎认为它只能表示规范化的查询。但实际不是这样的，相反，它可以表示任何规范的关系表达式。这些表达式反过来可以发挥很多作用，包括查询，但不仅仅局限于查询。下面再强调几点重要的事情：

- 定义一个元组集合来实现插入、删除，或者对关系变量进行修改（或者说，定义一个元组集合，并把它赋值给某个关系变量；
- 定义完整性约束（虽然这里讨论的关系表达式只是布尔表达式的一个子表达式，而且在 **Tutorial D** 中，一个 IS_EMPTY 调用不是不可以变化的）；

---

1 注意：前面的章节中没有一个例子涉及分支或循环。

- 定义安全性约束（看下面的进一步讨论）；
- 定义视图（同样看下面的进一步讨论）；
- 作为对优化和数据库设计领域进一步研究的基础[1]；

……

7.6.1 安全性

就像在第 1 章看到的，安全性约束（或者安全性控制）用来保证用户请求的合法性。换句话说，用户请求被限制为（a）允许执行用户请求的那些操作；（b）用户只允许访问他们被授权访问的数据库的部分。实际上，说明哪些情况是允许要比说明哪些情况是不允许的要更容易些。因此，一般支持定义安全性语言，即不是安全性约束，而是授权，这从效果来看与安全性约束是相反的（即授权某些事情，而不是限制某些事情）。下面给出几个例子加以说明[2]：

```
AUTHORITY SX1
    GRANT RETRIEVE { SNO , SNAME } , DELETE
    ON    S
    WHERE CITY = 'London'
    TO    Jim , Fred , Mary ;

AUTHORITY SX2
    GRANT RETRIEVE , UPDATE { STATUS , CITY }
    ON    S
    TO    Dan , Misha ;
```

对于安全性的扩展讨论（例如，授权以外的事情）可以在我的著作 *An Introduction to Database Systems*（第 8 版，Addison-Wesley，2004 年出版）中找到。

7.6.2 视图

考虑下面的查询（“获得伦敦供应商的信息”）：

```
S WHERE CITY = 'London';
```

假设特定的查询要反复多次地重复使用，那么我们就可以通过定义一个虚拟关系变量或视图来简化一下。

---

1 请注意，优化和数据库设计自身都具有非常丰富的标识。但是在两种情况下，为研究它们所付出的许多努力将不是很简单地具有某种可能性，除了关系模型中提供的稳固的逻辑框架外（进一步的讨论请参见第 9 章 。

2 包含了一些非安全性特征，但是这些例子被看作是语言的一种表示方式。SQL 包含了安全性特征，但是这些说明远远超出了本书的范畴。

```
VAR LS VIRTUAL ( S WHERE CITY = 'London' ) ;
```

现在整个查询就可以表示成如下的简化形式：

```
LS
```

或者如果我们还可以在原始的查询基础上再定义一个稍微复杂的变量，即“查询状态值为 10 的伦敦供应商的信息”：

```
LS WHERE STATUS = 10
```

我们详细分析一下这个例子。其核心思想就是当我们定义一个视图时，DBMS 通过把它存在一个称为“目录（catalog）”[1]的地方来记住它的定义，那么，当它处理查询时，DBMS就会在目录中查找视图的定义，然后用这个定义替代这个查询中引用视图的地方，即：

```
( S WHERE CITY = 'London' ) WHERE STATUS = 10
```

这个表达式的判断方式与关系表达式的判断方式完全相同。

因为这个简单的例子足以说明视图机制可以允许用户执行想要进行的查询，就好像视图是一个常规的关系变量，因而，对于用户来说在某些方面使用起来就要简化得多（这与高级语言中的宏变量非常相似）。再强调下面几点。

- 第一，因为关系的闭包特性要求视图必须精确地处理查询！这是因为它是那个属性的逻辑结果，如果（a）关系变量引用 *R* 允许出现在某个关系表达式 *rx* 的某个地方；（b）引用 *R* 是被赋予类型 *RT* 的一个关系变量，那么判断与 *RT* 具有相同类型关系的表达式就被允许出现 *rx* 中，而 *rx* 出现在关系变量引用 *R* 的某个地方。因而，作为这个规则的一个特定实例，我们可以通过视图定义表达式（例如：S WHERE CITY = ‘London’），在表达式 LS WHERE STATUS = 10 替代视图引用 LS。
- 第二，回顾第 1 章的图 1.2（数据库系统架构）。可以在该架构中增加一层，如图 7.1 所示。

回顾我在第 1 章所说的逻辑数据库与物理数据库的分离，目的是要允许用户实现数据独立性。但现在我不想更确切地解释这个事情。特别是，这种分离（即逻辑数据库与物理数据库的分离）允许我们实现数据的物理独立性，这就意味着我们可以改变数据的存储方式，对于用户来说访问时不需要对访问方式做任何的改变。通过定义视图实现视图与关系变量的分离，也允许我们实现数据的逻辑独立性，即我

---

1 对于给定的数据库，目录就是一组定义的系统关系变量集合，也被保存在数据库中，用来描述所要讨论的数据库。描述信息（有时也称为元数据）包括给定关系变量的属性和码的详细信息，以及完整性约束和授权的详细信息。

们可以改变数据的逻辑存储方式，对于用户来说访问时不需要对访问方式做任何的改变。

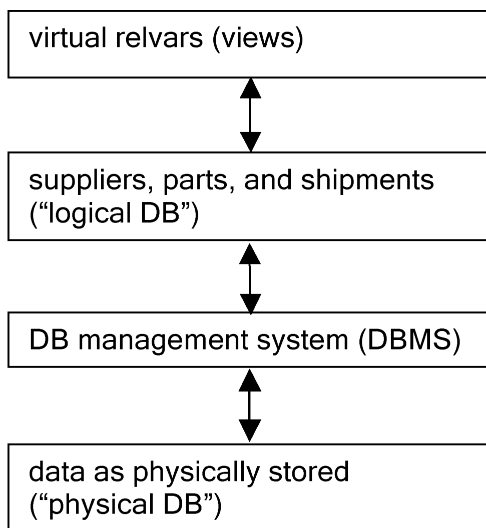


图 7.1 带有视图的数据库系统架构

对于视图的进一步讨论可以参见 *SQL and Relational Theory*，或者我的另外一本书，即 *View Updating and Relational Theory:How to Update Views*（2003 年，O'Relly 出版）。

7.7 结论

本书的第一部分到此结束。该部分详细介绍了关系模型本身，以及对于 DBMS 的重要性。对比而言，在第三部分，我们将看到标准的"关系型"语言 SQL，专门描述了（或许我应该说在某种程度上）如何使用该语言来实现我们所讨论的各种各样的关系特征。然而，在此之前，我们需要检验一些特定的话题（例如事务和数据库设计），就像我在前言中提到的那样，（a）为了评价数据库的全部功能，你确实需要对数据库有一个基本的了解；（b）但是不需要对关系模型本身有过多的了解（虽然它们都是建立模型的基础）。这就是下面将要介绍的第二部分的目的。

第二部分

事务和数据库设计

第 8 章 事务

因为时光，你也可以失而复得。

——Geoffrey Chaucer：*Troilus and Criseyde*（1385）

事务的概念只是与大多数用户有关，但这不完全是正确的。如果你是一些编写数据库应用程序的程序员，那么你肯定需要知道事务的确实含义，才能帮助你去编写代码，但是你也不必知道得过于详细。如果你是一个交互的用户，你可能根本不需要知道那么多。另一方面（前面我或多或少地提到过），如果你想要对数据库技术的全貌有一个概括的了解，那就必须对事务管理的细节有一个基本的认识。

前面我也曾经说过，事务管理本身并不是真正的关系型的话题。然而，有趣的是在特定的关系型环境中[1]，都要对事务管理进行基础研究。也就是说，提出这个概念是为了建立一个健全的科学研究基础。

8.1 什么是事务

事务就是程序的一次执行过程，或者是程序的一部分，它由一组逻辑工作单元组成。以 BEGIN TRANSACTION 语句开头，表示事务开始，以 COMMIT 语句或 ROLLBACK 语句结尾，表示事务执行结束，具体如下：

- BEGIN TRANSACTION 语句必须存在，显式说明；
- COMMIT 表示事务成功结束，显式说明；

---

1 参见由 K. P. Eswaran、J. N.Gray、R. A. Lorie 和 I. L. Traiger（1976 年 11 月举办的 CACM 19 会议，第 11 篇文章）撰写的基础文章 *The Notions of Consistency and Predicate Locks in a Data Base System*，以及 Jim Gray and Andreas Reuter 的标准文章 *Transaction Processing: Concepts and Techniques*（1993 年，Morgan Kaufmann 出版）。

- ROLLBACK 表示事务执行失败，事务回滚，通常隐式说明。

因而，任意给定的一个程序都包含一个依次执行的事务序列 [1]（序列中事务的数量也可以为 1），如 图 8.1 所示。

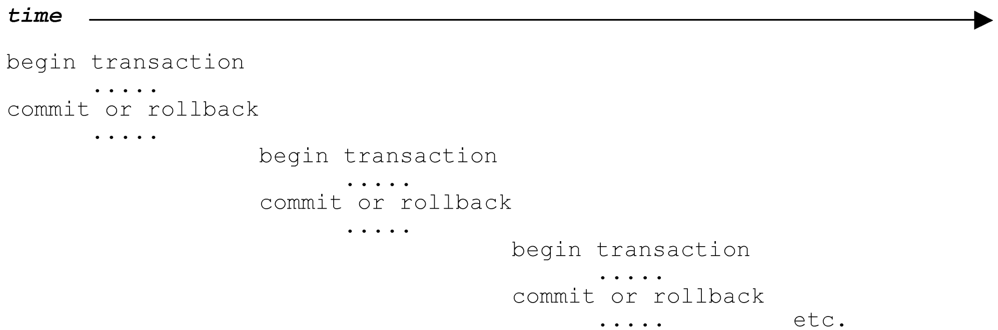


图 8.1 程序的事务执行序列

8.2 恢复

数据库中所有的修改操作都可以通过执行事务来实现，理解这一点非常重要。这是因为事务不仅仅是一般工作单元，它们也是一个恢复（*recovery*）单元。我来解释一下，通常以某类银行业务作为一个标准例子，下面的例子实现了从一个账户转账 100 美元到另一个账户，这是事务的伪代码：

```
BEGIN TRANSACTION ;
   UPDATE account 123 : { BALANCE := BALANCE - 100 } ;
   IF error THEN ROLLBACK ;
   UPDATE account 321 : { BALANCE := BALANCE + 100 } ;
   IF error THEN ROLLBACK ;
COMMIT ;
```

可以看到，这是一个原子操作（即从账户 123 转账 100 美元到另一个账户 321），事实上，它涉及对数据库进行两次单独的修改 [2]。而且，在两次修改操作之间，数据库实际上是处于不正确的状态，因此这时它就不能反映真实世界中的真正状态。显然，在真实世界中，两个账户的转账业务不能反映该账户中的总额，但是在这个例子中，100 美元只是在两次修改之间暂时丢失了。因而，一个事务中的逻辑工作

1 另一种说法为：如果当前正在执行一个事务，则不允许使用 BEGIN TRANSCTION。

2 由于可以使用多重赋值，实际上在 **Tutorial D** 中只是一次修改（参见第 6 章）。为了当前讨论方便，要必须假设这个运算符是不可用的（或者是不能使用的）。实际上，有必要显式声明一下，当在事务中使用基本理论时，不可能考虑多重赋值。

单元不能只单单涉及一次修改。相反，这个逻辑单元通常要涉及多个操作的序列，这个序列的目的是把数据库从一个正确的状态转换到另一个正确的状态，而且不必考虑任何中间点的正确性。

注意：现在进一步明确一点，该例子中数据库在两次修改之间的状态肯定是不正确的，是不一致的。但它不会破坏任何已知的完整性约束 1。我在第 6 章曾经提到过，正确性就意味着一致性，但是反过来是不一定的：不正确性并不意味着不一致性。因而，事务的目的是把数据库从当前的一致性状态转换到另一种一致性状态，而不用考虑任何中间点的正确性，这种说法更准确一些。在 8.2.2 节“ACID特性”部分我还要继续讨论这个问题。

显然，在该例中执行第一次修改而不执行第二次修改，这是不允许发生的，因为这会把数据库置于不正确的状态。理想的情况是我们想要一个严格的保证，即两次修改都能成功。但不幸的是，提供这种保证几乎是不可能的。因为操作就会出错，而且会在最可能出错的时候出错。例如，在两次修改之间会发生系统冲突，可能在第二次修改时发生运算溢出。但是正确的事务处理会提供次级最好的保证。特别是，如果事务执行一些修改，那么在事务到达预期的结束点前发生失败的话，事务管理就会提供一些保证，保证这些失败的事务会回滚（例如，undone 操作）。因而，事务要么全部执行，要么全部被撤销（就好像事务根本没执行一样）。在这种情况下，从外部来看非原子的操作序列就可以被看作是原子的。COMMIT 和 ROLLBACK 操作是实现这个保证的关键。

- 成功结束：COMMIT 可以保证事务修改后正常提交（这些修改被写入数据库中）。
- 不成功结束：ROLLBACK 可以保证把事务回滚到事务的开始执行状态（因此就会撤销事务的修改），然后终止事务。
- 隐式 ROLLBACK：美元转账例子中的代码包含了错误的显式测试，如果检测一个错误条件，就会迫使执行一个显式的 ROLLBACK。但是显然我们不能假设，也不能做任何假设，事务一直会包含对所有可能发生的错误的显式检测。因此，系统迫使因某种原因没有达到预期结束的失败事务执行隐式的 ROLLBACK，这里“预期的结束点”含义是显式的 COMMIT 或显式的 ROLLBACK。而且，如果发生系统冲突（这时即使事务本身没错，也会失败），那么系统就会重启，并让事务再重新执行一次。

因此，在这个例子中，如果事务成功执行了两次修改，就是执行 COMMIT，把这些修改写入数据库。如果任一个修改发生错误，都会执行 ROLLBACK，撤销目前所做的所有修改。如果发生了一个完全未预料到的错误，如系统冲突，那么系

---

1 而且，我们所写的完整性约束条件会被美元转账例子破坏，这是没有任何理由的（正确吗？）。

统就被迫执行 ROLLBACK。

8.2.1　恢复日志

你可能正在考虑，撤销修改可以实现吗？基本的解决方案是，系统会永久保存一个日志（通常在硬盘上），日志中记录了修改的详细信息，特别记录每次修改对象之前的值及修改对象之后的值，有时称为修改前的映像（*before-images*）和修改后的映像（*after-images*）[1]。因此，如果要撤销特定的修改，日志就变得非常重要了，系统要使用相应的日志记录把修改对象恢复到初始值（也就是说，把修改对象恢复到它修改之前的值）。当然，为了完成此过程，给定修改的日志记录必须在这个修改被实际写入到数据库之前记录到日志文件中[2]。这个协议被称为事先写入日志规则（*the write ahead log rule*）。

附加：当然，这个事先写入日志规则所提供的优点要比你期望得多。更特殊的是，这个规则需要实现（a）对于给定事务的所有其他日志都要在该事务的 COMMIT 记录被物理地写入日志之前，被物理地写入到日志中；（b）直到该事务的 COMMIT 记录被物理地写入到日志中，COMMIT 处理才能完成。

遵守以上规则的事务不仅仅是一个工作单元，也是一个恢复单元。下面我们讨论下一个话题，即事务的 ACID 特性。

8.2.2　ACID 特性

ACID 是缩写形式，代表原子性、一致性、隔离性、持久性。每个事务都具有这 4 个特性。

- 原子性（Atomicity）：事务要么全部执行，要么全都不执行。
- 一致性（Consistency）：事务把数据库从一个一致性的状态转换到另一个一致性的状态，而不需要去考虑保持任何中间点的一致性。
- 隔离性（Isolation）：任何事务的修改都与所有其他事务进行隔离，直到该事务被成功提交。
- 永久性（Durability）：一旦事务被成功提交，所有的修改会被永久保存在数据库中，即使后来发生了系统冲突。

原子性和持久性的含义是事务分别是一个工作单元和一个恢复单元。而且，一致性的含义是事务也是完整性单元（请看下面一段的说明），隔离性的含义是事务

1　你可以把“对象”看作是元组。虽然在实际情况中，“对象”更可能被认为是物理对象，比如磁盘页。

2　这里“被物理地写入”的含义是讨论的记录不仅仅被停留在内存的某个地方，而且在之后的某个时间也被写入到外存，起到永久保存的作用。

也是一个并发单元。现在我们已经知道了原子性、持久性的实现方法，但其他两种特性还需要进一步说明。

实际上我不想对一致性解释太多，而且提出这个特性我是有些犹豫的，因为它假设了直到事务正常提交时才能进行完整性约束检测（即“推迟检测”）。同时，在标准的SQL和特定的商业DBMS中，完整性检测的推迟是能真正实现的，坚持推迟检测的这个事实从逻辑上来说是不正确的（我在第 6 章曾简单地提到了，详细内容可以参照*SQL and Relational Theory*一书）[1]。就像我们在第 6 章看到的，这是因为关系模型需要立即检测所有的约束条件，这就暗示了只要涉及关系模型，那么完整性单元就不能是事务，只能是语句。

现在就剩下隔离性了。这里要利用两个部分详细地解释一下隔离性。

8.3 并发性

到目前为止，本章大部分的时间都假设在给定的时间系统中只运行了一个事务。但是现在要假设同时并发运行两个或多个事务，那么，就像第 1 章指出的，控制是必须的，这是为了要保证这些事务相互之间不能干扰（我们通常要假设事务之间是相互独立的）。例如，考虑图 8.2 所示的运行方案。该图的含义如下：首先，事务 *TX1* 修改对象 *p*，然后事务 *TX2* 要查询同一个对象 *p*，最后，事务 *TX1* 被回滚。这时事务 *TX2* 已经看到，它的操作要依赖于一个从没有发生的修改。

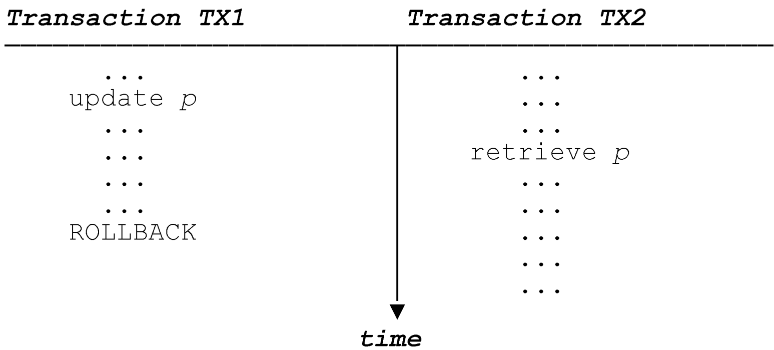


图 8.2 事务 *TX2* 依赖于一个未提交的修改

图 8.3 所示的运行方案更糟糕，事务*TX1* 和*TX2* 同时检索同一个对象*p*，接着 事务*TX1* 修改该对象p，然后事务*TX2* 也修改同一个对象*p*，因而*TX1* 的修改被覆盖了

1 因此，有意思的是，标识符“C”通常不用来代表一致性，而是代表正确性。

（在此时间点，*TX1* 的修改被称为“丢失”）[1]。

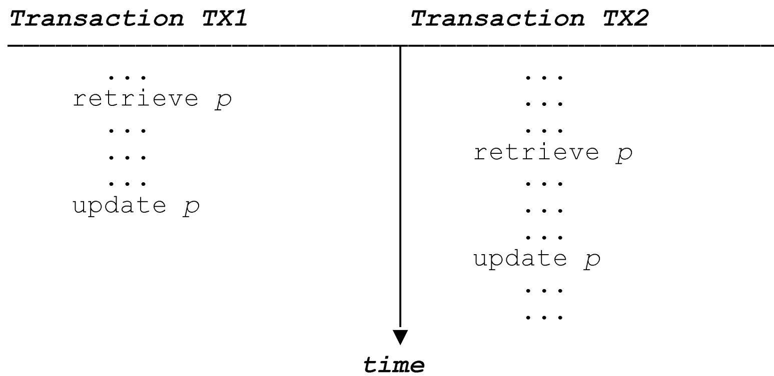


图 8.3 事务 *TX1* 的修改丢失

图 8.2 和图 8.3 发生的情况说明了一个事务的执行会受到另一个事务的干扰，并行的事务会产生一个完全不正确的结果。顺便说一下，如果允许事务并行执行，加以适当的控制，这些图中说明的问题（图 8.2 中依赖于未提交的修改，图 8.3 中的丢失修改）就不会发生了。然而，它们可能是最容易理解的例子。但关键的一点是我们确实需要这些控制。同时控制的类型在实际应用中也是可以见到的，被称为锁（locking）（虽然不仅仅有这一种可能的类型）。

8.4 锁

锁的基本实现原理很简单，事务 *TX1* 需要一种保证，它所感兴趣的对象不能被修改，直到事务 *TX1* 执行完毕，因此需要在该对象上加锁（以某种可以解释的方式）。获得锁之后造成的影响就是把其他事务隔离在讨论的对象之外（同样，以某种可以解释的方式），引入要特别阻止它们去修改该对象。*TX1* 能够继续它在某一知识领域的处理而其他任何事务都不能修改该对象，至少等到 *TX1* 释放锁（当然不能等到 *TX1* 完全执行结束）。

通常系统中支持两种类型的锁，共享锁（S 锁，也称为读锁）和排他锁（X 锁，也称为写锁）。不严格地讲，S 锁是可以相互兼容的，但 X 锁不能与其他锁兼容。举例如下，*TX1* 和 *TX2* 是不同的、但可以并行执行事务，那么：

- 只要 *TX1* 在对象 *p* 上持有 X 锁，*TX2* 在对象 *p* 上申请任何类型锁的请求都不会被批准。

---

1 这样的情况在实际中不会是不知道的。在乘坐飞机时，你碰到过你的座位被别人占用的情况吗？

- 只要 *TX1* 在对象 *p* 上持有 S 锁，*TX2* 在对象 *p* 上申请 X 锁的请求被拒绝，但可以申请 S 锁。

因而，在任意给定的时间内，可以同时有很多事务在对象 *p* 上持有 S 锁，但是至多只能有一个事务在对象 *p* 上持有 X 锁，在后面的例子中，没有其他的事务在对象 *p* 上持有任何类型的锁。

系统使用上述描述的机制目的是强化协议，保证不会发生图 8.2 和图 8.3 中提到的问题。下面解释了该协议的机制（但超出了本书的范围）[1]。

- 事务在对象 *p* 上的检索请求就暗示着在对象 *p* 上加 S 锁。
- 事务在对象 *p* 上的修改请求就暗示着在对象 *p* 上加 X 锁。注意：如果讨论的事务已经在对象 *p* 上加了 S 锁（在实际情况中这是非常可能的），那么隐式的加锁的请求就将 S 锁升级为 X 锁。
- 在这两种情况下，如果隐式的锁请求没有被批准（因为一些其他的事务对该对象持有冲突的锁），那么发出请求的事务就处于等待状态，直到冲突的锁被释放。
- 最后，COMMIT 和 ROLLBACK 都会使所持有的锁直到事务结束时才被释放。

现在我们来看看前面的协议是如何解决图 8.2 和图 8.3 中说明的问题的。图 8.4 是图 8.2 的一个修改版本，表示了遵守该协议下的事务 *TX1* 和 *TX2* 的交叉执行过程。首先，*TX1* 要修改对象 p，则获得对象 *p* 的 X 锁，接着 *TX2* 试图检索同一个对象 *p*。然而，*TX2* 在对象 *p* 上加 S 锁的隐式请求不被批准，所以 *TX2* 处于等待状态。然后，*TX1* 被回滚，X 锁被释放。现在 *TX2* 退出等待状态，获得对象 *p* 的 S 锁。但是 *p* 的值是 *TX1* 运行之前的值，所以 *TX2* 不再依赖于一个未提交的修改。

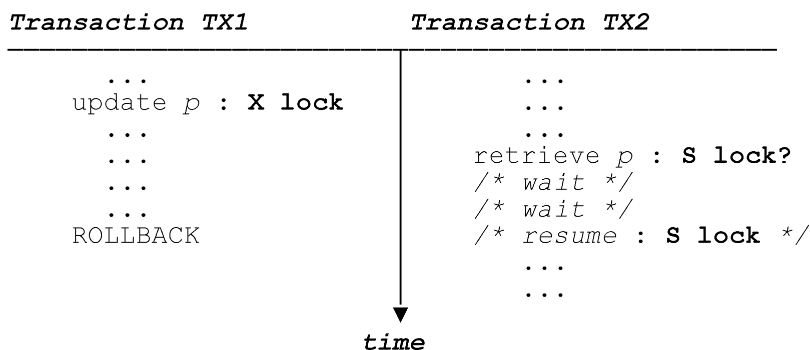


图 8.4　事务 *TX2* 不再依赖于一个未提交的修改

1　该协议的正式名称为"严格的两段锁协议"。实际上，真正的系统中一般采用各种各样的机制去改进这个协议。然而，此部分的描述已经远远超出了本书的范围。

回头引用一下 ACID 特性，看看这个锁协议是如何隔离图 8.4 中两个事务的。

现在转向图 8.5，它对图 8.3 进行了简单地修改。首先，*TX1* 检索对象 *p*，则获得对象 *p* 的 S 锁，接着 *TX2* 检索同一个对象 p，也获得该对象的 S 锁。*TX1* 试图修改 *p*，这个请求会发出一个加 X 锁的请求，但未被批准，*TX1* 则处于等待状态。然后 *TX2* 也试图修改 *p*，相似的理由也会使 *TX2* 处于等待状态，现在两个事务都不能继续执行了，所以不会产生任何修改的丢失。另一方面，也没有正在执行有用的工作。换句话说，我们通过把它转化为另一个问题的方式解决了丢失修改问题（但至少解决了最初的问题）。产生的新问题称为死锁（deadlock）。

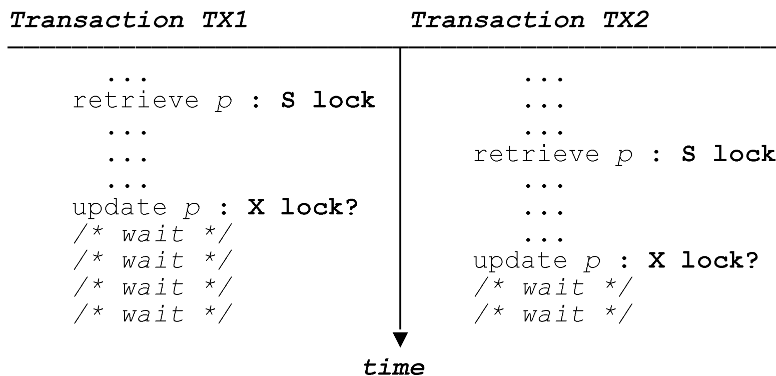


图 8.5 没有丢失的修改，但产生了死锁

对于死锁问题典型的解决方案为（a）选择一个死锁的事务，作为“牺牲者”，将其回滚（通常选择最年轻的事务，即最近执行的事务；（b）作为一个新的事务，重启“牺牲者”。

8.5 SQL 的讨论

因为我不想在后续的章节中过多地讨论事务（可以参见本书第三部分），在结束本章时，我想谈谈事务中经常用于SQL系统的一些事情（或多或少地）。SQL确实支持显式的BEGIN TRANSACTION、COMMIT和ROLLBACK语句（**Tutorial D**中也是如此）[1]，它对锁本身没有显式的依赖，即没有显式的语法来申请和释放锁。这就意味着系统可以自由使用除了锁以外的其他机制，这作为并发控制的基础。（在**Tutorial D**中也是如此。）但是SQL确实还具有其他的特征、概念和语句可以处理事务 （a）**Tutorial D**中是不具有的；（b）这些也远远超出了本书介绍的范围。这类

1 实际上，在 SQL 中是用 START TRANSACTION 来代替 BEGIN TRANSACTION。

特征的一个简单例子就是它具有事务隐式开始的例子，不需要执行显式的BEGIN TRANSACNTION语句，但是这样的特征还有很多很多。

8.6 练习

（出自于2004年Addison-Wesley出版的*An Introduction to Database Systems*，第8版。）术语调度（*schedule*）通常用于多个事务交叉执行时的检索和修改操作序列（例如，并发），下面的调度表示了事务 *T1*，*T2*，…，*T12*（*a*，*b*，…，*h* 都是数据库中的对象）操作的一个调度序列，如图8.6所示。

```
time t00     ..........
time t01   (T1)    : RETRIEVE a ;
time t02   (T2)    : RETRIEVE b ;
   ...     (T1)    : RETRIEVE c ;
   ...     (T4)    : RETRIEVE d ;
   ...     (T5)    : RETRIEVE a ;
   ...     (T2)    : RETRIEVE e ;
   ...     (T2)    : UPDATE e ;
   ...     (T3)    : RETRIEVE f ;
   ...     (T2)    : RETRIEVE f ;
   ...     (T5)    : UPDATE a ;
   ...     (T1)    : COMMIT ;
   ...     (T6)    : RETRIEVE a ;
   ...     (T5)    : ROLLBACK ;
   ...     (T6)    : RETRIEVE c ;
   ...     (T6)    : UPDATE c ;
   ...     (T7)    : RETRIEVE g ;
   ...     (T8)    : RETRIEVE h ;
   ...     (T9)    : RETRIEVE g ;
   ...     (T9)    : UPDATE g ;
   ...     (T8)    : RETRIEVE e ;
   ...     (T7)    : COMMIT ;
   ...     (T9)    : RETRIEVE h ;
   ...     (T3)    : RETRIEVE g ;
   ...     (T10)   : RETRIEVE a ;
   ...     (T9)    : UPDATE h ;
   ...     (T6)    : COMMIT ;
   ...     (T11)   : RETRIEVE c ;
   ...     (T12)   : RETRIEVE d ;
   ...     (T12)   : RETRIEVE c ;
   ...     (T2)    : UPDATE f ;
   ...     (T11)   : UPDATE c ;
   ...     (T12)   : RETRIEVE a ;
   ...     (T10)   : UPDATE a ;
   ...     (T12)   : UPDATE d ;
   ...     (T4)    : RETRIEVE g ;
time t36     ..........
```

图8.6 事务交叉执行的调度序列

假设 RETRIEVE *p* 获得对象 *p* 的 S 锁（如果成功的话），把 UPDATE *p* 锁升级为 X 锁（如果成功的话）。同时也假设所有的锁一直持有到事务结束。在时间点 *t36*，会有哪些事务处于等待状态，请画出等待图。此时有死锁存在吗？

8.7　答案

在时间点 *t36*，根本没有一个事务在做有用的工作！事务 *T1*、*T5*、*T6* 和 *T7* 已经执行结束（*T1*、*T6* 和 *T7* 成功结束，*T5* 不成功结束）。存在一个死锁，涉及的事务有 *T2*、*T3*、*T9* 和 *T8*。此外，*T4* 在等待 *T9*，*T12* 在等待 *T4*，*T10* 和 *T11* 都在等待 *T12*。我们可以通过图形的方式表示这种情况（称为等待图），在图 8.7 中，（a）顶点代表事务；（b）从顶点 *Ti* 到顶点 *Tj* 的有向边表示 *Ti* 等待 *Tj*；（c）从顶点 *Ti* 到顶点 *Tj* 的边采用数据库操作对象的名称和 *Ti* 等待的锁的类型进行标识（看图 8.7）。图中的环路代表发生了死锁。

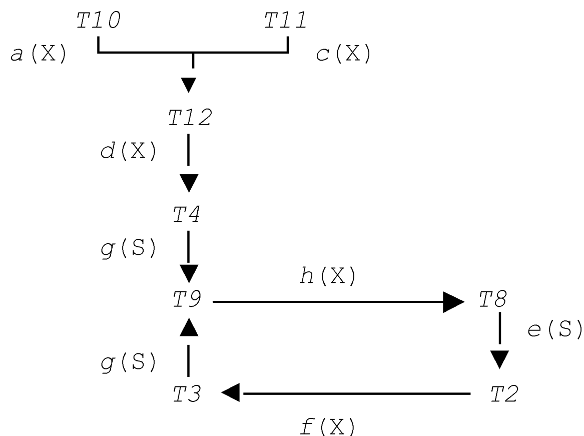


图 8.7　等待图

第9章
数据库设计

一般情况下，我确实不做设计工作。

——Anon：*Where Bugs Go*

就像第 7 章说明的那样，Codd 对关系模型的介绍将研究人员带入数据库管理的很多方面，其中一个方面就是数据库设计。实际上，设计理论是一个内容相当丰富的领域，其自身具有很丰富的文化内容。这个理论本身讨论如下问题：一个“好”的数据库设计到底应该由哪些部分组成？一个“好”的设计应该如何实现？当然，我们只是从表面描绘了这些事情，但是，与事务一样（第 8 章的主题），对数据库设计（或数据库设计理论）的熟悉程度是基本理解数据库技术内容的一个重要因素。

所以设计理论通常是关系理论中的一个固定组成部分。然而，它不是关系模型本身的一部分。相反，它是建立在这个模型顶层的独立理论。实际上，关系模型本身并不关心数据库设计得“好”与“坏”（**Tutorial D**中也是如此，更不用说SQL也是如此了）：只要数据库至少是关系级的，就要遵守信息理论，那么关系模型中的所有概念就仍然要使用，特别是关系代数中的所有运算符仍然要发挥作用。因而，如果数据库设计得不好，是用户要遇到困难，而不是关系模型本身[1]。

顺便说一下，我们一直使用的供应商-零件数据库就设计的很好（它具有三个关系变量 S、P 和 SP），我希望您也有同感。实际上，这是基本常识。很多设计理论只是通过某种方式被美化成基本常识，这是相当不公平的。说它是规范化的基本常识会更好一些。规范化是很重要的，因为如果某件事情被规范化了，就会有固定的机制。换句话说，我们可以采用这种机制进行工作。但是对这一点继续讨论的话，又会离我们要讨论的范围太远。如果你想进一步研究，可以参照我的另一本书

1 实际上 DBMS 受到的破坏更多，虽然被破坏的程度较小。

Database Design and Relational Theory: Normal Forms and All That Jazz（2012 年，O'Reilly 出版）。

9.1 无损分解

回顾第 2 章曾提到的关系一直是规范化的（例如，首先必须是第一范式，即 1NF），即所讨论的关系中的每个元组都必须与特定的标题一致。现在我们来看看这个定义如何在关系中使用的，但我们要把它扩展到关系变量中。

定义：关系变量 *R* 是 1NF，当且仅当可以合法赋值给 *R* 的每个关系 *r* 都是 1NF 的。当且仅当可以合法赋值给 *R* 的每个关系 *r* 中的每个元组应该满足（a）每个属性都恰好只有一个；（b）除此之外，没有其他条件。

当然，满足这个定义的每个关系变量都是 1NF 的。但是这个定义只提供了一个建立关系的基本条件。也就是说，我们可以定义一系列的较高级别的范式，如：2NF、3NF 等，这样，具有较高级别范式的关系变量就具有 1NF 中没有定义的属性。

我们来具体看一个例子，它只是 1NF 的，而不是更高级别的范式。图 9.1 给出了一个 SPCT 的关系变量的样本值，其属性为 SNO、PNO、QTY、CITY 和 STATUS，其属性的含义与前面相同（但是注意 CITY 在这里指的是供应商所在的城市），关系断言如下：

Supplier SNO supplies part PNO in quantity QTY, has status STATUS, and is located in

city CITY

SPCT

SNO	PNO	QTY	CITY	STATUS
S1	P1	300	London	20
S1	P2	200	London	20
S1	P3	400	London	20
S1	P4	200	London	20
S1	P5	100	London	20
S1	P6	100	London	20
S2	P1	300	Paris	10
S2	P2	400	Paris	10
S3	P2	200	Paris	30
S4	P2	200	London	20
S4	P4	300	London	20
S4	P5	400	London	20

图 9.1 关系变量 SPCT——样本数据

从图中就可以看出这个设计是多么的糟糕，很多数据都是冗余的，即很多的信息都被重复记录了多次。特别是，具有特定城市的供应商以及具有特定状态值的供应商出现了多次。而且，这些冗余的数据会直接或间接地导致“修改异常”。

- 插入异常：我们插入位于雅典的供应商 S5，直到该供应商提供零件。
- 删除异常：如果我们只删除供应商 S3，就会丢失所在城市为巴黎的供应商信息（即删除了过多的信息）。
- 修改异常：如果我们把 SNO 为 S1、PNO 为 P1 的元组中供应商所在城市从伦敦改为罗马，但供应商 S1 所在的其他元组值不变，我们会明显发现存在不一致性。注意，如果有完整性约束条件保证每个供应商必须位于一个城市的话，我们不能保证这个修改可以执行。但至少现在假设这个约束条件是不存在的。下一节中我将详细介绍这个问题。

再重复一次，前面的数据库设计是极其糟糕的[1]。解决这个问题的答案也很明显：我们需要把关系变量SPCT替换成两个单独的关系变量，一个就是常用的供应商变量S（为了简化，通常忽略供应商名字）[2]，另一个是供应关系变量SP。如图 9.2 所示，经过这样的改变后，就删除了数据冗余，因而避免了修改异常。

S

SNO	STATUS	CITY
S1	20	London
S2	10	Paris
S3	30	Paris
S4	20	London
S5	30	Athens

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

图 9.2　关系变量 S（简化后）和 SP（样本数据）

可以看出，分解出来的两个关系变量S和SP都是关系变量SPCT的投影[3]，而且，

1 公平地说，至少容易看到它糟糕的一部分原因是这个例子相当简单。在实际应用中发现冗余和异常可能是不太容易的。

2 在本章中我将继续省略供应商的名字，直到特殊说明。

3 除了投影关系 S 中的供应商 S5 的元组，它没有出现在 SPCT 的元组中（这是为了特殊说明分解是如何避免插入异常的）。

如果我们把这些投影再联接在一起，就要恢复成原来的关系变量。因而，从这个例子可以得出如下结论（或者至少是强烈建议的）：更高级别的范式（如 2NF、3NF 等等）应该满足（a）可以通过投影的方式来分解一个关系变量；（b）可以消除冗余；（c）分解前的关系变量应该等价于这些投影的联接。因为要特别考虑到（c），所以分解过程必须是无损的（当然，在这个过程中我们不能丢失任何信息是非常重要的）。同时，就像我们已经看到的，分解之后的结果是更高级别范式的关系变量，这个过程被认为是进一步的规范化，或者简称为规范化。另外当我们讨论分解过程时，因为（a）和（c）要被放在一起，所以我们通常说投影就是分解运算符，联接就是相应的分解运算符。因此，设计理论（或者我们这里讨论的设计理论的一部分）主要依赖于关系模型中的特定特征，尤其是依赖于投影和联接运算符，这在前面章节曾经提到过。

术语说明：严格地讲，术语投影和联接在前面的段落中必须加上引号。从本书第一部分可以看到，这是因为这些运算符实际上是应用于关系而不是关系变量。当然，我们也可以这样说“*R* 是 *R1* 和 *R2* 的联接”，这里的 *R*、*R1* 和 *R2* 是关系变量，但我们这样说的意思通常是指 *R* 的当前值关系 *r* 等价于 *r1* 和 *r2* 的联接，这里 *r1* 是 *R1* 的当前值，*r2* 是 *R2* 的当前值。但是，现在我们采用这种说法与其本身的含义稍有不同。特别是在规范化的情况下，当我们说 *R* 是 *R1* 和 *R2* 的联接时，只有在任一特定时间下 *R* 的当前值关系 *r* 等价于 *r1* 和 *r2* 的联接（这里 *r1* 是 *R1* 的当前值，*r2* 是 *R2* 的当前值）时，才保证一直取值为真（如果你感觉有些糊涂的话，我真的要说声抱歉。但有时明确这些说法是非常重要的）。

9.2　函数依赖

在前面的章节中我曾经提到，必须满足完整性约束条件是指每个供应商都只能位于一个城市。规范地讲，约束是函数依赖的一种。函数依赖是一个相当重要的概念（它们通常被描述为“不是非常基础的，但接近于基础的概念）。这里给出定义：

定义： *X* 和 *Y* 是关系变量 *R* 两个属性子集，那么函数依赖（FD）X→Y 在 *R* 中成立，当且仅当如果属性集合 *X* 中每个属性的值构成的集合唯一地决定了属性集合 *Y* 中每个属性的值构成的集合，则属性集合 *Y* 函数依赖于属性集合 *X*，记为：*X*→*Y*。属性集合 *X* 中的属性有时也称作函数依赖 *X*→*Y* 的决定因素（*determinant*），*Y* 称为被决定因素（dependent）。

举个例子，函数依赖 FD {SNO}→ {CITY}来自于关系变量 SPCT（一个供应商号码可以找到唯一对应的城市）。当然，关系变量 SPCT 中也存在函数依赖 {SNO} → {STATUS}，因此，我们可以把这两个函数依赖进行合并简化，如下：

```
{ SNO } → { CITY , STATUS }
```

注意，这里使用的是大括号。X和Y都是R的子集，因此也是集合，即使在{SNO} → {CITY}的情况下，它们也是单独的集合。以此类推，X和Y的值也都是元组，即使恰好有时它们的度为 1[1]。如果这些注释给了您提醒，也是因为这些情况与您现在所熟悉的情况是具有相似性的，例如，关键字是属性的集合，关键字值也是元组（如果需要唤醒记忆的话，请参见第 3 章练习 3.2 的答案）。

下面讨论码。假设关系变量 R 的码为 K，那么就存在函数依赖 FD：$K \rightarrow \{ A \}$，对于 R 中的每个属性都成立。即如果 R 中的两个元组具有相同的 K 值，那么它们必须是相同的元组，它们也肯定具有相同的属性值。另一种表示方式如下：

$$K \rightarrow X$$

这个函数依赖对于 R 的所有子集 X 都成立。因而，除了码之外的属性也存在函数依赖，但函数依赖中的决定因素都是码。不严格地讲，如果存在任何其他的函数依赖，那么这个设计就很糟糕。观察关系变量 SPCT，我们可以写出一些非码的函数依赖，如：{SNO} →{CITY}、{SNO} → {STATUS}，但{SNO}不是码。

最小函数依赖

在讲更高一级范式之前我需要再介绍一个概念。如果关系变量 R 中存在函数依赖 $X \rightarrow Y$，那么就存在函数依赖 $X' \rightarrow Y'$，X' 是 X 的子集，Y' 是 Y 的子集。换句话说，你可以向决定因素中添加属性，或者从被决定因素中减少属性，仍然可以得到关系变量中的函数依赖。例如，关系变量 SPCT 中存在函数依赖{ SNO }→{ CITY , STATUS }，则函数依赖{ SNO , PNO } → { CITY }也是成立的（向决定因素中增加了属性 PNO，从被决定因素中减少了属性 STATUS。那么，如果函数依赖 $X' \rightarrow Y'$ 成立，但函数依赖 $X \rightarrow Y'$ 对于 X' 任意的子集 X 都不成立，那么 $X' \rightarrow Y'$ 就是最小函数依赖。例如，关系变量 SPCT 中的函数依赖{ SNO , PNO } → { QTY }就是最小函数依赖，但{ SNO , PNO } → { CITY }不是（因为在该关系变量中存在函数依赖{SNO} → {CITY}）。

9.3 第二范式

下面给出 2NF 的确切定义。

定义：关系变量 R 是第二范式（2NF），当且仅当 R 中的每个码 K 和 R 的每个

1 回顾第 2 章练习 2.7 的答案，其中详细解释了元组的度的含义。

非码属性 A，其函数依赖 $K \to \{A\}$是不能简化的。注意，关系变量 R 的非码属性是指 R 的属性不是码的组成部分。

现在我可以来明确解释一下关系变量 SPCT 为什么是 1NF（因为它所有的关系变量都是 1NF）而不是 2NF 了。原因如下，（a）{SNO,PNO}是码；（b）存在函数依赖：{SNO ,PNO} → {CITY}、{SNO ,PNO} → {STATUS} ，但是（c）这个函数依赖是可以进行简化的。因此在关系变量 SPCT 中存在冗余的资源。

既然 SPCT 不是 2NF 的，要进一步规范化就需要采用投影的方式对它进行分解（当然是采用无损方式进行分解）。这里有一个很重要的公理，即 Heath 公理，可以有助于你对这个问题的理解。

> **Heath 公理**：关系变量 R 具有标题（即属性）H，X、Y、Z 是 H 的子集，X、Y、Z 的并集与 H 相等。XY 表示 X、Y 的并集，XZ 表示 X、Z 的并集，如果 R 中存在函数依赖 $X \to Y$，那么 R 就等于其投影 XY 和 XZ 的联接，它可以无损地分解成这些投影。

作为该公理的一个特例，如果 R 具有属性集$\{A,B,C\}$（本身都是单独的属性，来代替属性集合），R 中存在函数依赖$\{A\} \to \{B\}$，那么该公理告诉我们，R 可以被无损地分为 $R1$ 和 $R2$，具体如下：

$R1$ { A , B }，{ A }是码；

$R2$ { A , C }，{ A }是外码，$R1$ 是参照关系。

下面给出 2NF 的另一个定义来结束本节。可以证明这个定义与前面的定义是等价的（虽然我省略了证明过程），但有时这个定义更有用。

定义：关系变量 R 是 2NF，当且仅当 R 中存在的每一个非平凡的函数依赖 $X \to Y$，至少满足如下条件之一：（a）X 是超码；（b）Y 是子码；（c）X 不是子码。

这个定义需要再解释一下！

第一，函数依赖是平凡的，当且仅当它是永远成立的（即它不可能不成立）。在关系变量 SPCT 中，下面的函数依赖都是平凡的：

```
{ SNO , PNO } → { SNO }
{ SNO }       → { SNO }
```

事实上，很容易看到，如果函数依赖是平凡的，当且仅当函数依赖的右边（依赖因素）是左边的子集（决定因素）。

第二，关系变量 R 的超码是 R 标题的子集 SK，R 的码必须具有唯一性，但不必是经过简化的（参见第 3 章）。注意，所有的码都是超码，但是大多数超码不一定都是码（不是码的超码是超码的真子集）。另外还要注意，如果 SK 是 R 的超码，那么 R 中就需要存在函数依赖 $SK \to \{A\}$。练习：关系变量 P 中有多少个超码？（答案：16。这个统计中包含了码{PNO}和整个标题。注意，关系变量 R 的标题总是 R 的超码。）

第三，关系变量 *R* 的子码是 *R* 的码的子集。因而，所有的码都是子码，但大多数子码不是码（不是码的子码是子码的真子集）。练习：供应关系变量 *SP* 有多少个子码？（答案：4。这个统计包括了码{SNO，PNO}和空集{}。注意，空集是任意关系变量 *R* 的子码。）

码的规格说明注释：假设关系变量 R 的定义中包含了对码{k}的规格说明，那么我们可以说这种规格说明的意思就是指 *K* 是 *R* 的码。然而，严格地讲，它真正的含义是 *K* 是 *R* 的超码！这一点表明，当系统能够也肯定会满足 KEY 规格说明中暗示的唯一性属性时，它就不能满足相应的不可简化性。比如，零件关系变量 P，因为我们知道了该关系变量的含义（例如，我们了解相应的断言），那么属性集合{PNO，CITY}就不具有不可简化性（虽然它具有唯一性）。再重复一下，我们知道，但系统是不知道的。所以如果我们用{PNO，CITY}代替{PNO}来作为码，对于该关系变量，系统就不能满足零件号本身的约束条件，这个约束条件与 PNO-CITY 组合相反，它是具有唯一值的。换句话说，当我们用 KEY{*K*}作为关系变量定义的一部分时，系统就可以保证{*K*}是超码，但不一定是码。

9.4　第三范式

现在通过另外一个例子来讨论第三范式（3NF）。假设供应商关系变量中还有一个函数依赖{CITY} → {STATUS}（如图 9.3 所示，（a）为了和新的函数依赖一致，修改了供应商 S2 的状态值；（b）重新恢复了供应商名字属性 SNAME）。

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	30	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

图 9.3　具有函数依赖{CITY} → {STATUS}的关系变量 S——样本值

经过修改后，它是 2NF 的，您可以自己检测一下。然而它不是 3NF 的（看下面的定义），因此你可以看到，仍然存在数据冗余（特别是，对于给定的城市，其状态值要重复出现多次）。

定义：关系变量 *R* 是 3NF 的，当且仅当 R 中的每个非平凡函数依赖 $X \rightarrow Y$ 满足以下两个条件之一：（a）*X* 是超码；（b）*Y* 是子码。

因此，可以看到图 9.3 给出的关系变量 S 不是 3NF 的，因为（a）存在函数依赖{CITY} → {STATUS}；（b）该函数依赖显然是非平凡的；（c）{CITY}不是超码、{STATUS}不是子码；（d）所以，该关系变量不是 3NF，其中存在着冗余的资源。

既然给定的关系不是 3NF 的，规范化原理就要求通过投影进行分解（当然是采用无损的方式），采用 Heath 公理，可以得到两个投影，一个是在 SNO、SNAME、CITY 上的投影，另一个是在 CITY 和 STATUS 上的投影。练习：检查这个分解是否（a）满足 Heath 公理的条件；（b）是无损的；（c）删除了冗余信息。

顺便说一下，如果把 3NF 的定义与 2NF 的第二个定义做比较，你会发现那个 3NF 的关系变量一定是 2NF（反过来是不一定的）。这种情况通常采用如下简化的方式说明：3NF 一定是 2NF。

9.5　BC 范式

最后，我们介绍一下BC范式（BCNF），它确实是函数依赖中一种范式 [1]。下面给出其定义。

定义：关系变量 R 是 BCNF 的，当且仅当 R 中的每一个非平凡函数依赖 $X \rightarrow Y$ 中，X 是超码。

首先要注意的是，BCNF 肯定是 3NF（但反过来不一定成立）。另外也可以发现，满足 BCNF 的函数依赖或者是平凡的（显然我们不能抛掉这些函数依赖），或者是来自超码的函数依赖（显然我们也不能抛掉这些依赖）。因为有些人喜欢说："每个事实都是关于码的事实，是整个码，除了码之外就什么都不存在了。" 我要强调一下，虽然这句话很有趣、很吸引人的，但这个非正规的属性不完全是对的，因为它假设了所有其他的事情都只有一个码。

下面举一个属于 3NF 但不属于 BCNF 的例子。将供应关系变量修改一下，称之为 SNP，它增加了一个属性 SNAME，代表了可接受的供应商的名字。另外假设供应商名字都是唯一的（例如，没有两个供应商在同一时刻具有相同的名字。注意，在本书其他地方我没有明确做出这种假设）。下面是 SNP 的样本数据，如图 9.4 所示。

SNO	SNAME	PNO	QTY
S1	Smith	P1	300
S1	Smith	P2	200
S1	Smith	P3	400
..	.....	..	...
S2	Jones	P1	300
S2	Jones	P2	400
..	.....	..	...

图 9.4　SNP 的样本数据

从图 9.4 可以看到，这个例子中仍存在冗余：供应商S1 的每个元组都告诉我们S1 的名字为Smith，供应商S2 的每个元组都告诉我们S2 的名字Jones等等。同样，Smith的每个元组都告诉我们Smith的供应商号是S1，Jones的每个元组告诉我们Jones的供应

1　公正地说，BCNF 应该称为第四范式（实际上，它几乎满足了第四范式），但不幸的是，到它被广泛理解以及评价其重要性时，另一种范式（该范式已经超出了本书的范围）已经被命名为第四范式了。

商号为S2 等等。因此，该关系变量不是BCNF的。首先，它有两个码，即{SNP，PNO}和{SNAME，PNO}[1]；第二，标题的每个子集（特别是{QTY}）都函数依赖于这两个码；然而，第三，存在函数依赖{SNO} → {SNAME}和{SNAME} → {SNO}，这些函数依赖肯定不是平凡的，函数依赖的决定因素也不是超码，因此不是BCNF的（但它是 3NF的，因为两个两种情况下的依赖因素都是子码）。

最后，我再重申一下规范化原理：如果关系变量 *R* 不是 BCNF，那么就采用投影技术进行分解。比如 SNP，下面的两种分解都可以实现此目的：

- 分别在{SNO，SNAME}和{SNO，PNO，QTY}上投影；
- 分别在{SNO，SNAME}和{SNAME，PNO，QTY}上投影。

9.6 结论

再重复一下，BCNF 也属于一种范式，因此，2NF、3NF 本身并不是非常重要，重要的是，它们是把关系规范化成 BCNF 的基础。换一种说法，BCNF 是实际数据库中最重要的一种范式，数据库设计人员通常都应该努力保证把所有的关系变量规范化为 BCNF。但也要注意，BCNF 从定义上来说比 2NF、3NF 简单，这是因为定义中没有提到不可简化的函数依赖、非码属性、子码等等。

既然是这样，为什么还要定义 2NF、3NF？为什么在本章还要提到它们呢？下面几点可以回答这个问题。

- 研究 2NF 和 3NF 可以帮助人们更好地、逐渐地理解更高一级范式，以及它们的设计理论（特别是函数依赖）。
- 研究 2NF 和 3NF 同样可以帮助人们了解设计理论发展的历史（为什么这些范式被顺序地称为“第二”、“第三”）。
- 最后，研究 2NF 和 3NF 也可以帮助我们找出关系变量不是 BCNF 时，错误在哪里？实际上，BCNF 规定：函数依赖必须是出自于对码的依赖。如何违反这个条件呢？简单一点说，基本上有 2 种方式：（a）可以有对真子码的函数依赖（在这种情况下，关系变量不是 2NF），或者（b）可以有对非码的函数依赖（这种情况下，关系变量不是 3NF）。

最后我再说明几点来结束本章的内容。

- BCNF 是函数依赖中的最终范式，但它肯定不是最终范式，因为还有几个更高一级的范式（至少有 6 个或 7 个）。然而，从实用的角度看，BCNF

---

1 这是因为表示样本数据时，没给出容易引起误解的下划线。它有两个码，而且没有更好的理由去说明它们哪一个可以作为主码，它们几乎是等价的。

肯定是最重要的一个。

- 规范化是很重要的。但是在设计理论中还要涉及更多的内容，就像在 *Database Design and Relational Theory: Normal Forms and All That Jazz*（2012 年，O'Reilly 出版）一书中提到的，我在本书前面章节中曾提到过。

9.7 练习

9.1 在本章内容中，我曾经提到，关系变量 SPCT（参见图 9.1）存在如下的插入异常：我们不能插入供应商 S5 的城市信息 Athens，直到该供应商提供了零件。当然，我们也不能插入供应商 S5 的状态信息 30，直到该供应商提供了零件。那么确切的原因是什么？

9.2 供应关系变量SP中有多少个函数依赖？哪些是平凡的函数依赖？哪些是不可简化的函数依赖？

9.3 就像在本章内容中强调的，一个函数依赖就是一种特定的完整性约束条件。请采用 **Tutorial D** 的 CONSTRAINT 语句表示关系变量 SNP 的函数依赖：{SNO} →{SNAME}、{SNAME} → {SNO}。可以参见本书 9.5 节 BCNF 部分。

9.4 根据你的工作环境，请按照如下要求给出几个例子：（a）关系变量不是 2NF；（b）关系变量是 3NF，但不是 2NF；（c）关系变量是 BCNF，但不是 3NF。

9.5 考虑本书 9.4 节“第三范式”部分讨论的供应商 S 的关系模式，该关系变量的修改异常有哪些？

9.6 （出自 2004 年，Addison-Wesley 出版的 *An Introduction to Database Systems* 第 8 版。）一个确定的数据库中包含公司部门和员工等信息，如下：

- 一个公司有若干部门组成；
- 每个部门有若干员工，若干项目，若干办公室；
- 每个员工有一段工作经历（多个员工就有多个）；
- 对于每一项工作，员工都有对应工资（雇用期间，这个工作会对应多个工资）；
- 每间办公室有若干电话。

要求设计的数据库包含如下信息。

- 部门：部门号（唯一的），部门预算，部门经理的员工号（唯一的）；
- 员工：员工号（唯一的），当前的项目号，办公室号码，电话号码，员工目前工作的名称（包括工作的日期以及该项工作对应的工资）；
- 每个项目：项目号，项目预算；
- 每间办公室：办公室号码（唯一的），面积，所有电话的号码（唯一的）。

请设计一组恰当的关系变量来表示这些信息，并说明函数依赖中存在的一些假设。

9.8 答案

9.1 因为没有零件号 *p* 和供应数量 *q*，*Supplier S5 supplies part p in quantity q, has status 30, and is located in city Athens* 就不是一个真的断言。换句话说，关系变量 SPCT 中供应商 S5 的断言没有被初始化，就不能被判断为真或假。

9.2 关系变量 SP 中包含了 31 个函数依赖。规范地说，这些函数依赖中存在闭包。但关系代数中还没有办法能够处理这些闭包。

```
{ SNO , PNO , QTY } → { SNO , PNO , QTY }
{ SNO , PNO , QTY } → { SNO , PNO }
{ SNO , PNO , QTY } → { SNO , QTY }
{ SNO , PNO , QTY } → { PNO , QTY }
{ SNO , PNO , QTY } → { SNO }
{ SNO , PNO , QTY } → { PNO }
{ SNO , PNO , QTY } → { QTY }
{ SNO , PNO , QTY } → { }

{ SNO , PNO }       → { SNO , PNO , QTY }
{ SNO , PNO }       → { SNO , PNO }
{ SNO , PNO }       → { SNO , QTY }
{ SNO , PNO }       → { PNO , QTY }
{ SNO , PNO }       → { SNO }
{ SNO , PNO }       → { PNO }
{ SNO , PNO }       → { QTY }
{ SNO , PNO }       → { }

{ SNO , QTY }       → { SNO , QTY }
{ SNO , QTY }       → { SNO }
{ SNO , QTY }       → { QTY }
{ SNO , QTY }       → { }

{ PNO , QTY }       → { PNO , QTY }
{ PNO , QTY }       → { PNO }
{ PNO , QTY }       → { QTY }
{ PNO , QTY }       → { }

{ SNO }             → { SNO }
{ SNO }             → { }

{ PNO }             → { PNO }
{ PNO }             → { }

{ QTY }             → { QTY }
{ QTY }             → { }

{ }                 → { }
```

非平凡的函数依赖只有 4 个：

```
{ SNO , PNO } → { SNO , PNO , QTY }
{ SNO , PNO } → { SNO , QTY }
{ SNO , PNO } → { PNO , QTY }
{ SNO , PNO } → { QTY }
```

只有下面 7 个函数是不可简化的：

```
{ SNO , PNO } → { SNO , PNO , QTY }
{ SNO , PNO } → { SNO , PNO }
{ SNO , PNO } → { SNO , QTY }
{ SNO , PNO } → { PNO , QTY }
{ SNO , PNO } → { QTY }

{ SNO , QTY } → { SNO , QTY }

{ PNO , QTY } → { PNO , QTY }

{ SNO }       → { SNO }

{ PNO }       → { PNO }

{ QTY }       → { QTY }

{ }           → { }
```

9.3
```
CONSTRAINT C9A COUNT ( SNP { SNO , SNAME } ) = COUNT ( SNP { SNO } ) ;
CONSTRAINT C9B COUNT ( SNP { SNO , SNAME } ) = COUNT ( SNP { SNAME } ) ;
```

9.4 答案略

9.5 插入异常：对于给定的城市，我们不能插入其状态信息，直到该城市有供应商入驻。

删除异常：如果我们删除了特定城市的元组，就会丢失该城市的状态信息。

更新异常：如果我们修改了特定城市其中一个元组的状态信息，而没有修改另一个，就会产生数据的不一致性。

（那么如何分解为 3NF，避免这种异常呢？）

附加练习：针对本章 9.5 节中“BCNF”部分的 SNP，按照该题的要求进行练习。

9.6 最重要的函数依赖如下：

```
{ DEPTNO }       → { DBUDGET , MGRNO }
{ MGRNO }        → { DEPTNO }
{ PROJNO }       → { PBUDGET , DEPTNO }
{ EMPNO }        → { PHONENO , PROJNO }
{ EMPNO , DATE } → { JOB , SALARY }
```

```
{ PHONENO }        → { OFCNO }
{ OFCNO }          → { AREA }
```

通过属性的名称就可以看出其含义。同时做出的假设如下：

- 没有一个员工同时担任多个部门的经理；
- 没有一个员工同时在多个部门工作；
- 没有一个员工同时担任多个项目；
- 没有一个员工同时在多间办公室工作；
- 没有一个员工同时有多个电话号码；
- 没有一个员工同时担任多份工作；
- 没有一个项目同时被分配给多个部门；
- 没有一间办公室同时被分配给多个部门；
- 部门号、员工号、项目号、办公室号、电话号码都是全球唯一的。

下面给出一种可能的BCNF关系变量[1]：

```
DEPT { DEPTNO , DBUDGET , MGRNO }
     KEY { DEPTNO }
     KEY { MGRNO }
     FOREIGN KEY { MGRNO } REFERENCES
             EMP { EMPNO } RENAME { EMPNO AS MGRNO }

EMP { EMPNO , PROJNO , PHONENO }
     KEY { EMPNO }
     FOREIGN KEY { PROJNO } REFERENCES PROJ
     FOREIGN KEY { PHONENO } REFERENCES PHONE

SALHIST { EMPNO , DATE , JOB , SALARY }
     KEY { EMPNO , DATE }
     FOREIGN KEY { EMPNO } REFERENCES EMP

PROJ { PROJNO , PBUDGET, DEPTNO }
     KEY { PROJNO }
     FOREIGN KEY { DEPTNO } REFERENCES DEPT

OFC { OFCNO , AREA , DEPTNO }
     KEY { OFCNO }
     FOREIGN KEY { DEPTNO } REFERENCES DEPT

PHONE { PHONENO , OFCNO }
     KEY { PHONENO }
     FOREIGN KEY { OFCNO } REFERENCES OFFICE
```

注意，虽然这些关系变量都是 BCNF 的，但它们仍然存在冗余。例如，特定的

---

1 尤其要注意关系变量 DEPT 中的 FOREIGN KEY 说明。

员工在特定的时间会有一份特定的工作，在关系变量 SALHIST 中就要出现 *n* 次。这里的 *n* 表示员工持有该份工作时在不同的时间获得的工资的次数。另外，PROJ 在 PROJNO 和 DEPTNO 上的投影也一直等价于这些相同属性对 EMP、PHONE、OFC 做联接运算后进行的投影。这就表明，虽然规范化到 BCNF 是我们所希望的，但它还不足以消除冗余。

第三部分

SQL

第 10 章

SQL 基本表

非自然起点中的自然顺序。

——Jane Austen：*Persuasion* (1818)

回顾第 1 章列出的本书的计划如下（至少第一部分和第三部分）：

- 首先，我想解释一下关系模型本身，我将采用设计过程中使用的语言 **Tutorial D** 来加以解释。
- 然后，我想解释一下关系模型中的术语和概念，主要是采用SQL语言，与实际应用一致的方式来解释[1]。（因此，请注意本书中我没有尽力覆盖到全部的SQL，相反，只是涉及了与关系相关的方面。这在前言中作为一个主要特点曾经提到过。）

另外，在第 1 章中我也声明过，因为是以前制定的计划，所以本书与大部分的书不一样，本书只是介绍了一些目前在市场上可用的、关系数据库支持的概论性知识。而且，本书中大部分章节也与大部分概论性书籍不同，只是介绍了 SQL，仅此而已。

我相信，先了解关系模型，再学习 SQL 要比反过来的过程容易一些。这么说的一部分原因是按其他方式来做的话，需要很多还未了解的知识（就像我在前言中说的，不了解的话就会遇到很多困难）。

10.1　发展历史

SQL语言最初是由IBM研究部在 20 世纪 70 年代提出的。除了IBM，关于SQL的第 1 篇论文发表在 1974 年 5 月 1 日～3 日举办的数据描述、访问和控制会议上，

1 SQL 官方读为“ess cue ell”，但在第 1 章中我注释了一下，大家经常听到的发音为“equel”。本书中采用的是官方读音，因此写为“an SQL table”（而不是“a SQL table”）。

作者为ACM SIGMOD工作组的Donald Chamberlin和Raymond Boyce[1]，论文题目为*SEQUEL:A Structured English Query Language*。（考虑到合法性[2]，名字在后面都被改为SQL，即结构化查询语言。当然，早期名字的"sequel"发音有时仍然会遇到。）有趣的是，SEQUEL（或SQL）是与关系演算和（也许程度会更浅）关系代数存在明显的不同，虽然它后来也相继混合了二者的一些特征。

IBM研究院首先实现了该语言作为System R的一个可选的、课程的工业原型的接口。后来IBM又相继在SQL/DS和DB2[3]中实现了该语言，它们都不是IBM的产品（它们都是在 20 世纪 80 年代常常被使用的），大部分是Oracle的产品（首先使用是在 1979 年，比 IBM 的产品使用的要早）。1986 年，美国国家标准协会（ANSI）开发该语言的标准版本，这与IBM专业用语非常接近，但有些不正规，名称为SQL/86。在 1987 年，国际标准化组织（ISO）采用 SQL/86 作为国际标准（ANSI 是美国组的成员之一）。后来这个国际标准又经过多次演化，如 SQL/89、SQL:1992、SQL:1999、SQL:2003、SQL:2008 等等，当前正在使用的版本是SQL：2011。下面是正式的引用地址：

> International Organization for Standardization (ISO), *Database Language SQL*, Document ISO/IEC 9075:2011.

请注意：除非有明确的声明，否则本书中对 SQL 的所有引用都应该按照以前标准的版本含义去理解。当然，你的里程是可以变化的，就像他们说的那样，没有任何一款商业产品在整个生命周期中支持这个标准，然而同时每个产品确实支持对于特定产品很重要的一些特征，但这根本不是标准的一部分。另一方面，因为我限制了只遵循 SQL 的核心特征，所以希望在你喜欢的 SQL 产品中至少支持这些特征也是有理由的。当然，要与标准相一致。

值得一提的是，这个官方标准已经变成了大量文档（或者一组文档）。首先，它由以下几部分组成：

第 1 部分：框架
第 2 部分：基础
第 3 部分：CLI
第 4 部分：PSM
第 9 部分：MED

---

1 事实上，就是 BCNF 中的 Boyce（参见第 9 章 。

2 再重复一下，SQL 原来代表的是"结构化查询语言"，但现在（至少是根据标准制定的）它只是一个名字了，不代表任何意义。

3 实际上 SQL/DS 是以 System R 中的代码为基础的。相反，DB2 真的是一个完全重新实现的版本。

第 10 部分：OLB

第 11 部分：信息模式

第 13 部分：Java

第 14 部分：XML

这里我不想对每个部分进行详细讲解，只想解释一下第 2 部分（它是语言的核心部分），该部分大约有 1500 页，包括几页对于“可能的问题”和“语言机遇”的介绍。这个部分留给您去自行理解其中事实和图表的重要性。

10.2 基本概念

SQL 中基本的数据架构本身不是关系型的，而是一张 SQL 基本表（带有行和列，用来替代元组和属性）。因此，供应商-零件数据库的 SQL 版由三张表组成。（注意：这里没有说要采用术语表（table）来表示一个 SQL 基本表，除非进行明确说明。）基本的操作结构是 *SELECT – FROM –WHERE* 表达式，用来从“旧”的表产生“新”的表。例如，给出一个规范的 SQL 查询，“查询状态值大于 10 的供应商号和所在城市信息”，形式如下：

```
SELECT SNO , CITY
FROM S
WHERE STATUS > 10
```

现在我们来仔细分析一下。

10.3 表的特性

回顾第 2 章，关系具有如下特性：

- 不包含重复的元组；
- 元组是无序的，从上到下排列；
- 属性是无序的，从左到右排列；
- 关系是规范化的。

在本书第一部分我们可以从多处看到关系属性的重要性。因此，SQL 表如何来满足这些特性呢？答案是：不太容易。我们将按次序一一分析。

关系中从不包含重复的元组：但是 SQL 表中通常包含重复的元组。事实上，默认情况下就是这样的，除非明确说明要消除重复的元组。因此，你能立刻看到 SQL 表与关系不是一回事。

关系中元组是无序的，从上到下排列：SQL 表中的行同样是无序的，从上到

下排列。所以至少在这一点上，SQL 和关系模型是一致的。

关系中属性是无序的，从左到右排列：但是SQL表中的列从左到右是有序的。换句话说，与关系相反，SQL表中会存在“第 1 列”、“第 2 列”等等。因此，在这一点上，SQL表和关系是不同的[1]。

关系总是规范化的。首先回顾一下规范化的定义，是指它属于 1NF，即关系中的每个元组中每个属性都恰好有一个值，再无其他条件。但是 SQL 表的行所对应的 1 列或多列可以不具有任何值。在这一点上，你又可以发现 SQL 表与关系是不同的。图 10.1 给出了供应商的 SQL 表，其中供应商 S3 就没有 STATUS 信息，供应商 S5 没有 CITY 信息。

S

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake		Paris
S4	Clark	20	London
S5	Adams	30	

图 10.1 供应商 S 的 SQL 表——带有空值

顺便提一下，虽然在图 10.1 中用空白表示该处没有数值，但相应的行在该处是没有空白的。相反，此处不包含任何东西。SQL 把这些位置称为“空值”。但想要清楚地解释 SQL 的空值是很困难的。如果你真的曾经试图这样做，你很快就会被扰乱的，不可避免地得出这样一条结论，这是没有任何逻辑意义的。所以，我从来不试图做出任何解释。因此，我采用下面简短的方式来进行解释。

- SQL的空值主要是用来处理现实世界中信息不完整的问题（这确实是真正存在的问题，让我来加速一下寻求解决的答案）。例如，在现实世界中，我们经常需要处理这样的事情“出现一个不认识的地址”、“被通知到的演讲者”等等，这些问题真的是一些实际的问题。然而不幸的是，SQL的空值并没有解决这些问题，而且把问题弄得更糟糕，因为它们把自己带入了一个另外的、混乱的、复杂的、困难的境地[2]。
- 在任何情况下，SQL的空值显然都会破坏关系模型的规定，只是因为空值

1 实际上我们可以说，SQL 表与关系就不是一回事，除非讨论的表中只有 1 列。

2 解决这个问题的几种方法的冗长叙述请参见 *Database Explorations: Essays on The Third Manifesto and Related Topics*（2010 年，Trafford 出版），作者是我和 Hugh Darwen。简短的讨论可以参见 *SQL and Relational Theory*，其中对由 SQL 空值所引起的一些复杂问题进行了描述。

不是一个值（由定义可以看出），因此具有空值的SQL表会破坏**信息原则**（*The Information Principle*）[1]。注意：实际上，有时SQL把空值看作是一个值，有时当成是一个值。换句话说，这是具有随机性的，但是大部分情况下空值是不被看作是一个值的。SQL频繁尝试按照它们的方式去处理问题，却显然说明了另一方面的问题，这种语言存在深刻而严重的缺陷。

对此问题的进一步说明已经超出了本书的范围，讲解这些已经足够了（就像在 *SQL and Relational Theory* 一书中的解释一样），因为你从不会相信来自于具有空值的 SQL 数据库的答案。我把这种状态看作是一种精彩的表演。所以我的建议是，不要使用 SQL 的空值，也不要使用任何与它们相关的 SQL 特性（在本书中，我完全忽略了这一点，除非因为环境因素被迫才顺便提到它们）。

更多术语

如你所知，在关系模型中把关系值（relations）和关系变量（relvars）明显区分开。在 SQL 中也必须要存在类似的差别，但是相似的术语并没有进行区分。也就是说，SQL 没有采用“table value”和“table variable”的形式来区分这些术语。相反还采用了相同的术语，即表（*table*）（这其实是不太合适的），因此，必须要根据上下文环境对这个术语进行恰当的解释。

SQL中也没有与关系中的表头和表的内容相一致的术语。实际上，它没有一个恰当的表的类型的定义！[2]

10.4 修改表

SQL 中没有明确的表赋值运算符，但是它有 INSERT、DELETE、UPDATE 语句（没有 D_INSERT、I_INSERT 语句）。举个例子，把下面 SQL 中的部分与 **Tutorial D** 对应，这个例子在第 2 章曾经给出过（为了方便说明，我又重新给出了 **Tutorial D** 格式，前面是 **Tutorial D** 格式，后面是 SQL 格式）：

```
INSERT SP RELATION                                    /* Tutorial D */
      { TUPLE { SNO 'S5' , PNO 'P1' , QTY 250 } ,
        TUPLE { SNO 'S5' , PNO 'P3' , QTY 450 } } ;
```

1 而且具有重复的 SQL 表，以及按照从左到右顺序排列的 SQL 表等都会破坏信息原则，这里指的是所有的 SQL 表。（当然包括一个例外，就是只有一个列的 SQL 表，列的顺序也就不重要了）。

2 这里不要被误导，它调用“类型化后的表”来处理事情（参见 *SQL and Relational Theory*，可以看到详细的解释）。

```
INSERT INTO SP ( SNO , PNO , QTY )                  /* SQL */
       VALUES ('S5' , 'P1' , 250 ) ,
              ('S5' , 'P3' , 450 ) ;

DELETE SP WHERE QTY < 150 ;                         /* Tutorial D */

DELETE FROM SP WHERE QTY < 150 ;                    /* SQL */

UPDATE SP WHERE SNO = 'S2' : { QTY := 2 * QTY } ;   /* Tutorial D */

UPDATE SP                                           /* SQL */
SET    QTY := 2 * QTY
WHERE  SNO = 'S2';
```

10.5 等值比较

在第 1 章里我曾经提到过，与任何给定类型*T*有关联的、所支持的运算符必须要包含等值比较运算符，因为如果没有等值比较运算符，我们就不能解释给定的值*v*是否是特定集合*S*的元素[1]。我也曾经讲过，比较*v*1=*v*2 为真，当且仅当*v*1 和*v2* 具有相同的值（当然要具有相同的类型）。因此，如果某个操作符Op满足*Op*(*v1*) ≠*Op*(*v2*)，那么*v*1=*v*2 就为FALSE。那么，SQL如何来匹配这些需求呢？

首先，我们考虑当 *T* 是表类型的时候如何处理。我们已经看到，SQL 不支持类型，因此，更别说要支持与类型相关的等值比较运算符了。注意：实际上对于这个问题我们讨论的有点多了，对“表的比较”部分的进一步讨论将放在第 11 章进行。

如果 *T* 是标量类型，会发生什么情况呢？不幸的是，情景是相当复杂的。但是，我只能给出一个最简单的概括总结。首先，SQL 支持强制类型转换（例如隐式的类型转换），“=”就可以赋值为 TRUE，即使比较的是不同类型的数据。其次，特别是字符串类型，即使比较的数据具有相同类型但又明显存在不同时也可能赋值为TRUE（稍后我会给出一个例子）。另外，即使比较的数据没有差别，“=”也有可能不被赋值为 TRUE（这里针对的是所有类型，但是字符串类型除外）。特别是比较的数据都是空值时会发生这种情况，但不仅限于此种情况。此外，还有用 XML 类型定义的系统以及用户定义的类型，也根本没有定义“=”运算符。

我们对前面的内容进行简单说明，假设字符串变量X和Y的值分别为字符串“AB”和“AB ”（注意，第二个字符串后面有一个空格），这两个值是明显不同的。然后，在某种环境下，X=Y也可以为TRUE，即使CHAR_LENGTH (X) =CHAR_

1 我们甚至不能讨论给定的一个值 *v* 是否是类型 *T* 的某个值。

LENGTH (Y)不成立。（显然后者的比较数长度为 3，前者长度为 2。）但也要注意，即使X=Y被赋值为 TRUE，X| |X = Y| |Y也不成立[1]！我把对这种奇特行为的详细解释留给你去思考。以上论述足以说明，这种结果很不乐观，也许超出了你的预期。

最后，我来说说完整性。SQL 确实支持行类型的等值比较（也支持其他类型的比较）。然而，具体的细节已经超出了本书的范围，详细解释请参见 *SQL and Relational Theory*。

10.6 定义表

现在给出表 S、P 和 SP 的 SQL 定义（在 SQL 中，采用 CREATE TABLE 语句定义表）。首先给出表 S 的定义，如下：

```
CREATE TABLE S
     ( SNO VARCHAR(5) NOT NULL ,
       SNAME VARCHAR(25) NOT NULL ,
       STATUS INTEGER NOT NULL ,
       CITY VARCHAR(20) NOT NULL ,
       UNIQUE ( SNO ) ) ;
```

说明：

- 定义暗示了表 S 是一个基本表（与 **Tutorial D** 的基本关系变量相似）。注意：SQL 也支持视图，我确信这也是你期望的，但是详细的说明超出了本书的范围。
- 定义中的第 2～6 行说明表中的列以及赋予表的完整性约束。每一列都有名字、类型，但不允许有空值（由于特殊说明了 NOT NULL）。列的类型可以是 SQL 允许的任何类型。然而，要注意 SQL 不支持与 **Tutorial D** 的 CHAR 相似的类型（参见第 1 章），所以 SNO、SNAME、CITY 的类型为 VARCHAR（可变长度的字符串），即可以任意定义长度，但要注意随需的最大长度。当然，列的说明顺序也是很重要的，因为在表中通常按照从左到右的顺序来定义列。
- SQL 中的 UNIQUE（SNO）相当于 **Tutorial D** 的 KEY{SNO}。
- 默认情况下，初始的基本表都是空的（也可能说明一个非空的初始值，但详细的说明超出了本书的范围）。

强调：一定要注意，前面的 CREATE TABLE 语句从逻辑上来说相当于调用

---

1 同样 X LIKE Y 也不成立。换句话说，在 SQL 中，*v1* 和 *v2* 可以相等，但不能进行“like”运算。

TABLE 类型发生器，除此之外，没有其他作用（从语言逻辑上来说，没有规定的次序）。当你意识到 UNIQUE（SNO）（它的作用是用来定义完整性约束）没有紧跟在列定义后面但是可以出现在任何地方的时候，这个事实会越来越明显，例如，出现在 SNO 和 SANME 定义之间。不要在单独的列定义上提到 NOT NULL 说明，它也是用来定义完整性约束。换句话说，我以前也曾经提到过，SQL 不支持表类型发生器，它也是根本没有给出一个恰当的“表类型”的定义。

下面给出表 P 和 SP 的定义，然后再对其进行详细的解释：

```
CREATE TABLE P
     (  PNO VARCHAR(6) NOT NULL ,
        PNAME VARCHAR(25) NOT NULL ,
        COLOR VARCHAR(10) NOT NULL ,
        WEIGHT NUMERIC(5,1) NOT NULL ,
        CITY VARCHAR(20) NOT NULL ,
        UNIQUE ( PNO ) ) ;

CREATE TABLE SP
      ( SNO VARCHAR(5) NOT NULL ,
        PNO VARCHAR(6) NOT NULL ,
        QTY INTEGER } NOT NULL ,
        UNIQUE ( SNO , PNO ) ,
        FOREIGN KEY ( SNO ) REFERENCES S ( SNO ) ,
        FOREIGN KEY ( PNO ) REFERENCES P ( PNO ) ) ;
```

“表标识符”

SQL 没有使用术语“表标识符”（*table literal*），但它支持该结构，VALUES 表达式提供了与其同样的功能，例如：

```
VALUES ('S1' , 'Smith' , 20 , 'London' ) ,
       ('S2' , 'Jones' , 10 , 'Paris' ) ,
       ('S3' , 'Blake' , 30 , 'Paris' ) ,
       ('S4' , 'Clark' , 20 , 'London' ) ,
       ('S5' , 'Adams' , 30 , 'Athens' )
```

（事实上，我们以前在“修改表”部分的 INSERT 例子中见到过 VALUES 表达式的例子。）

顺便提一下，要注意前面 VALUES 表达式的定义要依赖于列中从左到右的顺序。也要注意，真正的表标识符应该被授权允许出现在表的表达式允许出现的任何地方，但是 SQL 中允许表的表达式可以在多处使用，但是 VALUES 表达式却不允许（参见第 14 章练习）。这就是我说这个表达式只提供了部分表标识符功能的原因。

10.7 SQL 系统与程序系统

我们再来回顾一下第 1 章图 1.3 的代码片段，用它来说明程序系统中各种熟悉的概念：语句、类型、变量、赋值、标识符、只读运算符（包括比较运算符）。再来回顾一下与数据库直接相关的所有概念，看看这些与 SQL 的说明到底相差多远。

- SQL 语句：我们已经看到了 SQL 中存在 INSERT、DELETE、UPDATE 语句，也存在 CREATE TABLE 语句。
- SQL 类型：我们已经看到各种 COLUMN 的类型，如：VARCHAR（*n*）、INTEGER、NUMBERIC（*p*，*q*），我也简要地提到了一些其他类型，包括用户定义的类型。不幸的是，我们也看到在 SQL 中表本身没有类型。
- 表变量：SQL 中存在表变量的定义，但没有使用此术语。
- 表赋值：支持 INSERT、DELETE、UPDATE 语句，但不支持表赋值本身。
- 表的标识符：参见前面部分中对于 VALUES 表达式的讨论。注意：我们也看到一些标量标识符的例子（例如，‘S1’，‘Smith’，20，‘London’），我们还看到了一些行标识符的例子，即带有括号的表达式（‘S1’，‘Smith’，20，‘London’）。
- 表的值：SQL 中存在该定义，但没有这个术语。在特定时间 S、P 和 SP 的值就是这样的值。该值也可以用 VALUES 表达式来说明。
- 表的只读运算符：虽然我们已经看到了一个例子，但仍需要详细讨论（参见第 11 章和第 12 章）。表达式 SELECT SNO, CITY FROM S WHERE STATUS>10 就是一个例子，该例子位于“基本概念”部分，采用 SQL 的语法来说明。
- 表的比较运算符：虽然以前提到过（参见“等值比较”部分），但仍将在第 11 章详细讨论。

10.8 练习

下面哪个说法是正确的？

10.1 SQL 表中的行没有顺序。

10.2 SQL 表中的列没有顺序。

10.3 SQL 表中不存在没有命名的列。

10.4 SQL 表中不存在两个或两个以上同名的列。

10.5 SQL 表中不包含重复的行。

10.6 SQL 表总是 1NF 的。

10.7 SQL 表中列定义可以是任意复杂的。

10.8 SQL 表本身有类型。

10.9 SQL 表具有指针类型的列。

10.9 答案

在给出详细的答案之前，我想提醒您注意一下前面提到的两点事实，即 SQL 和关系模型是一致的。我想让你从这种状态中得出自己的结论。

10.1 SQL 表中的行没有顺序。正确

10.2 SQL 表中的列没有顺序。错误

10.3 SQL 表中不存在没有命名的列。错误

在基本表中不存在任何没有命名的列，这是正确的（偶尔视图也会出现这种情况）。然而不幸的是，特定的表的表达式在实际中真的会产生没有名字的列。下面给出一个此种表达式的小例子以及对应的表中数据，如图 10.2 所示（当然，与我们通用的例子相似）。

```
SELECT PNO , 2 * WEIGHT
FROM P
```

注意：这个问题按照标准来说，没有名字的结果列应该有 DBMS 来假定给出一个名字，而且要与表中的名字有所区别，是没有定义过的。但是，只存在于部分标准中的这项规则也只是一种迟到的企图来修改已经被破坏的情况（实际上从一开始就已经被破坏了）。同时，也要注意到未定义的名字会随着数据库的变化而变化。而且，发现任意给定情况下的名字是一件非常不重要的事情。因此，实际上，不用去计较问题中的列是否有名字。

10.4 SQL 表中不存在两个或两个以上同名的列。错误

对于基本表来说不存在同名的列（视图也是如此），但是特定的表的表达式却可以产生同名的两列或多列（甚至具有同一类型）。下面给出一个小例子以及表中的数据，如图 10.3 所示。

```
SELECT PNO ,PNO
FROM P
```

注意：更常见的接近于现实的例子如下。

```
SELECT *
FROM S , P
```

这个例子给出了具有两个 CITY 列的表，详细讨论请参见第 11 章。

PNO	
P1	24.0
P2	34.0
P3	34.0
P4	28.0
P5	24.0
P6	38.0

图 10.2 没有列名字的表

PNO	PNO
P1	P1
P2	P2
P3	P3
P4	P4
P5	P5
P6	P6

图 10.3 具有同名的列

10.5 SQL 表中不包含重复的行。错误。

同样，在基本表中这个论断是正确的，UNIQUE 说明可以阻止出现重复的列，但是这个说明是可选的。详细讨论请参见第 11 章。

10.6 SQL 表总是 1NF 的。错误

只有表示关系的表才能说成是 1NF 的，但是 SQL 不总是这样。注意：只有基本的情况下才满足这个论断，因为 NOT NULL 说明至少可以保证每一行每一列都有值（而且是对应类型的值），但是这个说明是可选的。而且，特定的某种表的表达式也可以产生带有空值的结果，即使输入的时候没有空值（参见第 12 章给出的一个例子）。而且，在任何情况下，在商业数据库中，列的顺序都是从左到右的，也不会提到未命名列的可能性以及重复列的可能性。

10.7 SQL 表中列定义可以是任意复杂的。正确

10.8 SQL 表本身有类型。 错误

10.9 SQL 表具有指针类型的列。正确

在本章中我没有讨论这个问题。但是令人难过的是，SQL 表中真的允许某一列的值采用指针方式来指向某个目标表中的行。指针被称为引用值（***reference values***），引用该值的列类型为 *REF type*。坦率地说，SQL 表中包含这些特征的原因还没有完全弄清楚。但可以肯定的是，这些特征的有无似乎对于实现一些有用的功能没有多大影响。所以，虽然包含了这些特征，但不要使用它们！

第 11 章

SQL 操作符 I

你知道我的方法，并使用它们。

——Arthur Conan Doyle: *The Sign of Four* (1890)

现在我们开始来论证 SQL 是如何来支持关系代数中各种操作符的，或者支持到什么程度？由于各种各样的原因，本章结构故意采用与本书第一部分第 4 章平行的结构。然而，首先，要提醒你一下：

- SQL 表中存在重复的行，所以发现这种情况的时候你就要想办法消除（至少要把问题中的表按照关系型操作来处理）。
- SQL 表按照从左到右的顺序排列属性，但是写代码时可以不依赖于这种次序，因为代码不是关系型的。
- SQL 表中可以包含空值，但是不允许出现。

注意：在 *SQL and Relational Theory* 一书中描述了称为"从关系型角度使用 SQL"的规则，因此如果你遵守那本书中的推荐规则，那么你的所有处理行为几乎都证明 SQL 好像真的是关系型的，而且你会体会到真正的关系型系统（或至少是接近于关系型的）给你带来的好处。

11.1 限制

本章中大部分时间我计划使用与第 4 章同样的例子。所以下面的这个查询就是第 4 章用来介绍显示运算符的查询，即查询 1："查询零件重量小于 12.5 磅的零件信息"，为了进行比较，我首先重复描述一下 **Tutorial D** 的格式：

```
P WHERE WEIGHT < 12.5
```

SQL 查询的格式如下：

```
SELECT PNO , PNAME , COLOR , WEIGHT , CITY
FROM   P
WHERE  WEIGHT < 12.5
```

说明：

- 与 **Tutorial D** 不同，SQL 支持使用（.）作为引用符号，称为域操作符（*dot qualified references*）。实际上，上面给出的表达式只是一种简写形式（注意下面 SELECT 和 WHERE 子句中的（.）引用符号）。

```
SELECT P.PNO , P.PNAME , P.COLOR , P.WEIGHT , P.CITY
FROM   P
WHERE  P.WEIGHT < 12.5
```

上面的查询中把能够使用域操作符的地方都尽量使用了该操作符。

- SQL 也支持其他的简写形式，例如：

```
SELECT P.*
FROM   P
WHERE  P.WEIGHT < 12.5
```

SELECT 子句中的表达式“P.*”表示列出了 P 中所有的列，采用逗号来进行分隔，顺序为从左到右。然而，正是因为要依赖于从左到右的顺序，这种表示法就给粗心的人带来很多陷阱，所以在本书中我没有采用这种表示。如果使用的话，就会进行特殊说明。同时还要注意，域操作符（.）是可选的，也就是说，可以采用“SELECT *”代替“SELECT P.*”。

- 同 **Tutorial D** 一样，SQL 也允许在 WHERE 子句中使用布尔表达式来表示一些简单的限制条件，后面我们会看到一些例子。

11.2　投影

请看下面的查询，查询 2：“查询每个零件的编号、颜色和所在城市”。同样我首先采用第 4 章 **Tutorial D** 的格式来表示这个查询，然后采用 SQL 格式表示，具体如下：

```
P { PNO , COLOR , CITY }

SELECT PNO , COLOR , CITY
FROM   P
```

注意：SQL 中不具有 **Tutorial D** 的 ALL BUT 特性。

现在我们讲解相当重要的一点，即第 4 章 4.3 节投影部分第 2 个例子的查询，查询 3：“查询当前存在的零件颜色和城市”，首先给出 **Tutorial D** 格式：

```
P { COLOR , CITY }
```

SQL 的语法格式如下：

```
SELECT COLOR , CITY
FROM   P
```

查询结果如图 11.1 所示。

如你所见，这个结果不是一个关系，它包含了重复的元组，因此不能满足任何码的约束（注意观察图中没有双下划线的标志）。换句话说，与关系模型的投影操作不同，SQL 的 SELECT 表达式不能消除重复，除非进行显式说明要求消除重复。消除重复的方式为采用关键词 DISTINCT，将其放在 SELECT 子句的逗号列表之前，具体格式如下：

```
SELECT DISTINCT COLOR , CITY
FROM   P
```

运行结果如图 11.2 所示。

COLOR	CITY
Red	London
Red	London
Red	London
Green	Paris
Blue	Oslo
Blue	Paris

图 11.1　查询 3 的运行结果

COLOR	CITY
Red	London
Green	Paris
Blue	Oslo
Blue	Paris

图 11.2　查询 3 消除重复值的运行结果

说明几点：

- 上面第 1 个例子中采用 DISTINCT 说明并不是不正确，但是它不会有任何影响（为什么？）。通常情况下，省略 DISTINCT 意味着你在进行一些非关系型的处理，因此我要说：不能这样做。
- 在第 4 章中，为了说明关系代数的闭包特性，给出了一个同时包含了限制和投影的例子。查询 4：“查询零件重量小于 12.5 磅的零件号、颜色和所在城市”，下面给出 SQL 的语法格式（与第 10 章 10.2 节“基本概念”部分给出的 SELECT 表达式基本相似）：

```
SELECT DISTINCT PNO , COLOR , CITY    /* DISTINCT 可以省略 */
FROM   P
WHERE  WEIGHT < 12.5
```

11.3　并、交、差

考虑如下的查询，查询 5：“查询至少存在一位供应商或者至少提供一种零件的城市信息”。**Tutorial D** 的格式如下：

```
S { CITY } UNION P { CITY }
```

SQL 的格式同它类似：

```
SELECT CITY FROM S
UNION
SELECT CITY FROM P
```

奇怪的是，使用UNION后（和投影不同）[1]，SQL确实通过默认的方式消除了重复，但通过DISTINCT进行显示说明也不是错误的，例如：

```
SELECT CITY FROM S
UNION  DISTINCT
SELECT CITY FROM P
```

你甚至可以这样写：

```
SELECT DISTINCT CITY FROM S
UNION  DISTINCT
SELECT DISTINCT CITY FROM P
```

回顾关系模型，UNION 的操作数必须是同一种类型的（例如，具有同样的标题），结果也是和操作数具有同样类型。然而，SQL 表是没有类型的，所以不可能采用这个简单的规则，也确实是不能采用。事实上，关于 UNION 运算的操作数和运行结果的 SQL 规则是相当复杂的，复杂到不能在这里给出详细解释的程度。（当然，如此复杂的原因就是因为 SQL 没有表类型的定义。）不管怎样，目前也可以说，只要在 UNION 或 UNION DISTINCT 后面使用可选的关键字 CORRESPONDING 进行说明，SQL 规则就几乎退化为关系规则。更准确地说，如果表 *t1* 和 *t2* 同时具有列名为 *C1*，*C2*，…，*Cn* 的列（而且仅有这些列），如果列 *t1* 的列 *Ci* 和 *t2* 的列 *Ci* 具有同种类型（$i = 1, 2, \ldots, n$），则表达式如下：

```
SELECT C1 , C2 , ... , Cn FROM t1
UNION  [ DISTINCT ] CORRESPONDING
SELECT C1 , C2 , ... , Cn FROM t2
```

该表达式将产生没有重复行的表，而且列的名字和类型与*t1* 和*t2* 相同。实际上，只要使用CORRESPONDING进行说明，在表达式的两个SECLECT子句逗号列表中就不需要按照从左到右的顺序说明各列 [2]。但是如果你省略了CORRESPONDING，

1 阐明一下这个观点：在结果中不存在任何重复的行，即使在两个操作数表中存在相同的行，或者在任一表中存在相同的行。

2 就像这段论述中建议的（有些不太光明正大），如果不采用 CORRESPONDING 说明，那么当且仅当两个 SELECT 子句的列名逗号列表中具有相同的从左到右的顺序时，这些列才能被认为是一致的。然而要注意的是，即使你使用 CORRESPONDING 说明，结果中列的从左到右顺序也是按照两个逗

则所处理的就不是关系型的，所以我再次强调一下：不要省略。

至于交运算（INTERSECT）和差运算（EXCEPT），以上针对并运算讨论的内容完全适用于二者，因此，这里就不再举例说明。

规范特性

回顾第 4 章关系型的并操作符拥有的一些重要的规范特性：（1）交换性（*r*1 *UNION r*2 与 *r*2 *UNION r*1 等价）；（2）结合性（*r*1 *UNION r*2 *UNION r*3）与(*r*1 *UNION r*2) *UNION r*3 等价）；（3）自反性（*r* UNION *r* 简写为 *r*）。这些特性同样适用于关系型的交运算符。但是 SQL 与之类似的操作符通常不具有交换型（这是由于固定了从左到右的次序），也不具有自反性（这是由于存在重复的行）。然而，它们具有结合性。

11.4 更名

在第 4 章里，使用下面的查询解释 RENAME，即“查询至少一个供应商的名字或者至少一个零件的名字，并用 name 显示。”（就像我在那一章曾经说过的，这个例子是人造的，不是一个很自然的查询，但是它足以能够说明我要表达的观点。）**Tutorial D** 的格式如下：

```
 ( ( S RENAME { SNAME AS NAME } ) { NAME } )
   UNION
( ( P RENAME { PNAME AS NAME } ) { NAME } )
```

SQL 的语法格式如下：

```
SELECT SNAME AS NAME FROM S
UNION  CORRESPONDING
SELECT PNAME AS NAME FROM P
```

如你所见，SQL 实际上支持 RENAME 操作符，在 SELECT 子句的逗号列表中采用 AS 来加以说明。注意，在这个例子中我省略了 CORRESPONDING 关键词，这是因为表中只有一列，但是使用它也不是错误的。

11.5 练习 I

11.1 写出下列查询的 SQL 表达式：

号列表中的一个列表的列顺序显示的。换句话说，我们不能完全脱离 SQL 列的从左到右的顺序属性，但是写代码时不能依赖于它！

a．查询所有的供应关系。

b．查询供应商 S2 提供的零件号码。

c．查询状态值在 15 和 25 之间的供应商，消除重复值。

d．查询状态值低于供应商 S2 的状态值的供应商号码。

e．查询由巴黎的所有供应商供应的零件号码。

11.2　请根据自己的情况写出一些 SQL 查询表达式。最好利用自己的数据库，但如果没有，也可以根据供应商-零件数据库来构造查询。

11.6　答案 I

11.1　在第 4 章的练习 4.7 中给出了这些查询的 **Tutorial D** 表达式。

a.
```
SELECT SNO , PNO , QTY
FROM SP
```

b.
```
SELECT DISTINCT PNO
FROM   SP
WHERE  SNO = 'S2'
```

注意：这里不需要使用 DISTINCT，为什么？

c.
```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  STATUS >= 15 AND STATUS <= 25
```

d.
```
SELECT SNO
FROM   S
WHERE  STATUS <
     ( SELECT STATUS
       FROM S
       WHERE SNO = 'S2' )
```

与第 4 章的表达式一样，这个答案需要进一步解释。首先，我们已经说明了这一点，SQL与**Tutorial D**一样，WHERE子句中的布尔表达式通常要比简单的约束条件复杂得多。其次，跟在“<”符号之后的子表达式（由封装在括号中的SELECT-FROM-WHERE表达式组成）是SQL子查询[1]。至少从概念上来讲，首先计算子查询的值，返回如下形式的带有一行和一列的表[2]（见图 11.3）。

1 顺便说一句，可以把一个 SELECT-FROM-WHERE 表达式嵌套在另一个 SELECT-FROM-WHERE 表达式里（当然，可以嵌套任意层次），在 SQL（或者 SEQUEL）语言的最初名字中出现结构化也是合情合理的。

2 图中缺少双下划线，这并不是失误。详细讨论请参见《*SQL and Relational Theory*》。

STATUS
10

图 11.3 带有一行一列的表

现在发生了双重的强制转换（例如，隐式的类型转换）。表被强制转换为包含一行的表，反过来这一行又被强制转换为单标量值。（相反，**Tutorial D** 需要显式说明这些类型转换，这一点你可能回忆起来了。）标量值然后被插入到原来的查询里，获得如下查询：

```
SELECT SNO
FROM S
WHERE STATUS < 10
```

进而可以获得所希望的全部结果。

但是，有时也会发生一些例外，下面讨论几种情况。首先给出一个初始的表达式，但是这次采用显式的域（.）运算符来说明。

```
SELECT S.SNO
FROM   S
WHERE  S.STATUS <
      (  SELECT S.STATUS
         FROM S
         WHERE S.SNO = 'S2' )
```

我希望大家弄清楚，上面表达式中第 1 行和第 6 行对于“S.SNO”的两次列引用表达的不是同一个意思，同样，在第 3 行和第 4 行对于“S.STATUS”的两次列引用也是如此。事实上，SQL遵循块引用规则，虽然这些引用规则相当复杂，也超出了本书的预期。然而，我们可以在FROM子句中使用AS说明的形式来清晰地表示这种规则，同样可以采用域（.）运算符来说明 1：

```
SELECT X.SNO
FROM   S AS X
WHERE  X.STATUS <
     ( SELECT Y.STATUS
       FROM S AS Y
       WHERE Y.SNO = 'S2' )
```

上述修正格式中的符号 X 和 Y 就是一种域变量（***range variable***），它们是变量（从逻辑上来说，与编程语言中的变量不同），作用范围为表 S，在整个表达式求值时某个给定的时间点，它们表示了表中的某些行（当然表示的不是相同的行）。而

---

1 注意，以前讨论的 AS 子句（在列的更名部分）出现在了 SELECT 子句中，而不是 FROM 子句中。

在 **Tutorial D** 中不需要使用这种结构。实际上，域变量是关系运算（参见第 7 章或附录 D）具有的一种特征，而关系运算当然以关系代数为基础。

e.
```
SELECT PNO
FROM   P
WHERE  NOT EXISTS
     ( SELECT *
       FROM   S
       WHERE  CITY = 'Paris'
       AND    NOT EXISTS
            ( SELECT *
              FROM SP
              WHERE S.SNO = SP.SNO
              AND SP.PNO = P.PNO ) ) ;
```

这个练习（实际上与第 4 章的有些相似）放在这里有些不公平，因为我认为不能只用本书中这一部分讨论的内容来回答这个问题。然而我来简短地解释一下，(a) SQL表达式的EXISTS子查询的结果为真，当且仅当由子查询说明的表中至少包含 1 行[1]；(b) 整个表达式的两个子查询的变量是有相关性的（即相关子查询），这意味着它们包含了对于外层表达式中所定义条目的引用；(c) 我以前提过的与使用“SELECT”有关的问题没有出现在EXISTS表达式中，所以在这种环境下，使用该结构是很安全的。(顺便说一下，在SELECT子句的子查询中省略DISTINCT也是安全的，这个子查询表示了调用EXISTS的变量，准确地说，这是为什么呢？)

11.2 答案略

11.7 联接

SQL 有多种不同的方法或方式来表示联接操作。同样，使用第 4 章的联接查询，即“对于每一个供应关系，查询其零件号码、数量和相应供应商的详细信息”。**Tutorial D** 格式如下：

```
SP JOIN S
```

我想要讨论的第一种 SQL 格式与 **Tutorial D** 的格式非常相似：

```
SELECT * FROM SP NATURAL JOIN S
```

强调：

- 这种形式依赖于在两个操作表中具有同样名称和类型的列上进行联接运算来完成（**Tutorial D** 也是如此）。我强烈推荐采用这种形式来实现联接运

1 SQL 的 EXISTS 运算符与关系运算中的 EXISTS 等同。

算，其原因在 *SQL and Relational Theory* 中有详细讲解。

- 整个表达式产生的结果列如下：SNO – PNO – QTY – SNAME – STATUS – CITY（在SQL中总是按照从左到右的顺序来写）。通常情况下，SELECT * FROM *t1* NATURAL JOIN *t2* 的结果列中第一位是联接的列，然后按照次序依次为*t1* 的列，*t2* 的列[1]。
- 当然，同样要避免使用“SELECT *”，在本章中的前面部分也曾解释过。这里使用它的主要目的是可以看见森林中的树木，即将注意力集中在这部分的核心主题上，也就是联接运算。相反，推荐使用的格式如下：

```
SELECT SNO , PNO , QTY , SNAME , STATUS , CITY
FROM SP NATURAL JOIN S
```

但这个联接运算本身会有一些损失（即使在单独的行上来写出 SELECT 和 FROM 子句）。

11.7.1 另一种格式

当然，SQL 实际上不需要遵循引用上述的列的命名规则。我设想（？）的部分原因是它提供了很多种不同的表示前面那个例子的方法。下面就给出不同的方法，但是我不想去讨论具体的细节：

```
SELECT SP.SNO  /* 或者 S.SNO */ , PNO , QTY , SNAME , STATUS , CITY
FROM   SP JOIN S
ON     SP.SNO = S.SNO    /* 注意必须使用(.)运算符 */

SELECT SNO /* 不是 S.SNO 或者 SP.SNO! */ , PNO , QTY , SNAME , STATUS , CITY
FROM   SP JOIN S
USING ( SNO )

SELECT SP.SNO /* 或者 S.SNO */ , PNO , QTY , SNAME , STATUS , CITY
FROM   SP , S
WHERE  SP.SNO = S.SNO /*注意必须使用(.)运算符*/
```

11.7.2 规范特性

回顾第 4 章，关系型的联接运算符具有交换性、结合性、自反性。但相反，SQL 的联接运算通常不具有交换性（因为列按照从左到右排序），也不具有自反性（由于存在重复的行[2]）。然而，它具有结合性。

---

1 我在 *SQL and Relational Theory* 中写到，你开始看到了按照从左到右的顺序处理商业事务的痛苦了吗？

2 也由于存在空值。

11.7.3 笛卡儿乘积

看下面笛卡儿乘积的查询例子，即“查询所有当前供应商和零件号码对”。**Tutorial D** 的格式如下：

```
S { SNO } TIMES P { PNO }
```

至于 SQL，有两种不同的查询格式：

```
SELECT SNO , PNO
FROM S CROSS JOIN P
```

或者更简单一些的格式：

```
SELECT SNO , PNO
FROM   S ,   P
```

再考虑一下第 4 章的查询，即“查询供应商 SNO 没有提供的零件号 PNO 的有序对 SNO-PNO”。**Tutorial D** 的格式如下：

```
( S { SNO } TIMES P { PNO } ) MINUS SP { SNO , PNO }
```

SQL 的格式如下：

```
SELECT SNO , PNO
FROM   S ,   P
EXCEPT CORRESPONDING
SELECT SNO , PNO
FROM   SP
```

11.8 基本表表达式的求值

在 Codd 早期论文中曾给出定义，除了自然联接之外，还有一个称为“θ”联接的运算符，“θ”表示任何一种标量比较运算符（如“=”、“≠”、“<”等等）。现在 θ-联接不再是最初含义的运算符。事实上，这里定义为是乘积的比较运算（这就是为什么我没有在第 4 章讨论的原因）。下面是采用 SQL 格式给出的几个不等联接的例子，即供应商的城市与零件城市不相同（所以，在 SQL 中，θ 替换为“≠”或者“<>”）：

```
SELECT SNO , SNAME , STATUS , S.CITY AS SCITY ,
       PNO , PNAME , COLOR , WEIGHT , P.CITY AS PCITY
FROM   S , P
WHERE  S.CITY <> P.CITY
```

采用下面三个步骤来对表达式求值。

1．计算 FROM 子句，得到表 S 和 P 的乘积。注意，如果按照关系型的特点进

行操作，至少在计算成绩之前要把其中一个 CITY 进行改名。但 SQL 不会对它们改名，因为 SQL 表的列按照从左到右排序，因此可以通过初始的顺序来区分两个 CITY，为了简化，这里忽略了细节部分。

2．然后，计算 WHERE 子句，得到乘积的限制运算结果，即消除了两个城市值相等的行。注意，如果用“=”代替“≠”（或者“<>”），这个步骤就变为：保留两个城市值相等的行，这种情况称为等值联接。换句话说，等值联接也是一种 θ-联接，但 θ 为“=”。练习：等值联接与自然联接的区别？

3．最后，计算 SELECT 子句，得到这个限制的“投影结果”，投影的列采用 SELECT 子句说明。把“投影”加上引号，是因为最终结果消除了重复值，就像真正的投影运算一样，除非采用 DISTINCT 说明。（实际上，这个例子中也执行了一个改名操作，SELECT 提供的其他功能将在第 12 章中详细介绍，所以这里为了简化，忽略了细节部分。）

我们来做一个大致的对比，FROM 子句与乘积对应，WHERE 子句与限制运算对应，SELECT 子句与投影对应。因此，整个 SELECT-FROM-WHERE 表达式的含义就是在乘积的限制运算上进行投影。所以我只给出了一个不太严格，但相对合理的 SELECT-FROM-WHERE 表达式的语法定义，即我给出了计算这个表达式的一个概念性的算法。但现在没有任何迹象表明 DBMS 使用了这个算法来计算表达式的值，反之，它可以使用它喜欢的任何算法，只要保证所使用的算法能够得到与概念中定义的算法一致的结果。使用不同的算法也有很多益处（通常是从性能角度考虑），因此可以按照不同的顺序计算各个子句，或者重写原来的查询。然而，只有证明 DBMS 所使用的算法在逻辑上等价于定义的算法时，它才能任意使用算法。实际上，优化器的工作之一就是找到一个在概念上等价、性能更好的算法。（回顾第 1 章，优化器就是 DBMS 的构件之一，负责为如何实现用户的请求来进行决策。）

11.9 表的比较

第 4 章讨论了关系比较运算符“=”、“≠”、“⊆”（包含于）、“⊂”（真包含于）、“⊇”（包含）、“⊃”（真包含）。SQL 不直接支持这些运算符，但是可以使用变通的方法。例如，考虑下面的关系比较，采用 **Tutorial D** 格式如下：

```
S { SNO } ⊃ SP { SNO }
```

该表达式的含义为“查询没有供应零件的供应商信息”。SQL 的表示格式如下：

```
EXISTS ( SELECT SNO FROM S
         EXCEPT CORRESPONDING
```

```
          SELECT SNO FROM SP )
```

和

```
NOT EXISTS ( SELECT SNO FROM SP
             EXCEPT CORRESPONDING
             SELECT SNO FROM S )
```

解释如下：回顾联系 11.1e 的答案，SQL 表达式 EXISTS 子查询返回值为 TRUE，当且仅当由子查询说明的表至少包含一行。因此，在前面的表达式中，EXISTS 表明必须至少存在一个供应商位于 S 中，而不在 SP 中，NOT EXISTS 表明没有在 SP 中出现的供应商，而存在于 S 中。注意，如果要依赖于 SP 到 S 的外码约束条件，则可以省略 NOT EXISTS 部分。同样也可以省略两个 CORRESPONDING，而不会有任何损失，这是因为在这两种情况下，每个操作的表中都只有一列。但是包含的话，也不是错误的。

顺便说一下，SQL 中的 EXISTS 与 NOT EXISTS 分别与 **Tutorial D** 中的 IS_NOT_EMPTY 和 IS_EMPTY 对应。

对于表格等价性的说明：我曾经说过，SQL 不直接支持关系比较运算符。然而，“=”与“≠”是几乎直接支持的。下面我来解释一下。警告：这可能会有点复杂！

首先，如果我可以被允许使用这个术语的话，SQL表的内容不是一个集合，而是一个包（bag）（是专门由一些行组成的包）。（回顾第 5 章中包的概念，即包类似于集合，但允许存在重复的元素。）现在，SQL实际上确实支持包的概念（被允许使用的术语是多重集合），这些包包含了许多不同种类的元素[1]。具有若干的多个包只有一种特例。现在，SQL中包含若干行的一个包并不等同于SQL中的表，即使两种结构从表面上看有些相似，这主要是因为作用于SQL表的运算符不适用于SQL中包含行的包（为了简要讨论一下，所以我把它们放在前面提了一下）。然而，至少有可能实现其中一种结构向另一种结构的转化。

现在，在 SQL 的行包与 SQL 表之间重要的区别是行包的等值比较是直接支持的，而表的等值比较却不支持。所以如果我想检测两个表是否相等（如：表 S 在 CITY 上的投影与表 P 在 CITY 上的投影是否相等），就必须要把两个 SQL 表转化为行包，然后在行包上进行等值检测。

我希望你能理解到目前为止我所讲解的内容，但是当我告诉你把表转化为行包的运算符被称为 ***TABLE*** 的时候，可能又把你弄糊涂了。因而，为了检测表 S 和 P 在 CITY 上的投影是否相等，可以采用如下格式：

```
TABLE ( SELECT DISTINCT CITY FROM S ) =
TABLE ( SELECT DISTINCT CITY FROM P )
```

1 然而，在任意特定包中的元素都必须是同样类型的。

冒着再把你弄糊涂的风险，我告诉你一个事实，即有 TABLE 运算符产生的行包中的行是没有列名的。因而，前面例子中进行的行包实际上如图 11.4 所示。

```
┌──────────┐      ┌──────────┐
│          │      │          │
├──────────┤      ├──────────┤
│ London   │      │ London   │
│ Paris    │      │ Paris    │
│ Athens   │      │ Oslo     │
└──────────┘      └──────────┘
```

图 11.4　没有列名的行包

顺便说一下，例子中的 DISTINCT 都是必须使用的。例如，如果 London 出现在表 S 的 CITY 列 2 次，而在表 P 的 CITY 出现了 3 次（假设两个表中都不存在其他的 CITY 值），那么相应的投影就应该是等价的，但是相应的行包却不等价，因为没有消除重复的行。

11.10　显示结果

与关系中的元组一样，SQL表中的行从上到下是没有顺序的。但当表格显示在终端屏幕上时，这些行当然必须按照从上到下的顺序排列。而且，对于用户来说，这种顺序是有意义、有帮助的，而不是随意排列的（因为人类都习惯于按照某种特定意义的顺序去识别和处理各类事情）。正因为如此，SQL提供了ORDER BY运算符，是为了给表中的行排序 1，例如：

```
SELECT SNO , CITY
FROM   S
WHERE  CITY <> 'Athens'
ORDER  BY CITY
```

当然，在关系代数中没有与 ORDER BY 对应的运算符，这是因为 ORDER BY 不是关系运算符，更确切地说是因为它的结果不是关系型的。现在，请不要误解我了，我没有说 ORDER BY 没有用处，我要说的是它不能作为关系表达式的一部分出现。

事实上，ORDER BY 在其他方面也与关系代数的运算符有所区别，例如，它不是一个函数。在本书中描述的关系代数的所有运算符（事实上，应该用只读运算符这个术语表达会更容易理解）都是函数，这就意味着对任何给定的输入都只有一个可能的输出。相反，ORDER BY 针对同一个输入可能会产生几个不同的输出。因此，考虑上面的 ORDER BY 例子，可能会产生如下几种输出（为了简化，这里只给出了供应商号码，省略了其他信息）：

---

1 **Tutorial D** 支持类似的操作符，称为 ORDER。

- S1 , S4 , S3 , S5
- S1 , S4 , S5 , S3
- S4 , S1 , S3 , S5
- S4 , S1 , S5 , S3

对函数的解释：我曾经说过，本书中描述的关系代数的所有运算符都是函数。然而，我要提醒的是，这些运算符与 SQL 中对应的部分却大多都不是函数！这是因为(就像在第 10 章解释过的)即使 *v*1 和 *v*2 不同时,SQL 有时也认为比较式 *v*1=*v*2 的值为 TRUE。例如，考虑字符串“Paris”和“Paris ”(注意第 2 个字符串后面带有空格)，显然它们是不相等的，然而 SQL 有时却把它们看作是相等的。因此，按照标准，特定的 SQL 表达式被称为“可能的非确定性”。下面给出一个简单的例子：

```
SELECT DISTINCT CITY FROM S
```

如果某个供应商的 CITY 值为“Paris”，另一个为“Paris ”，那么结果中或者包含“Paris”，或者包含“Paris ”(也可能都包含)，但是我们会获得哪一种结果却是不确定的，可能今天获得一种结果，明天又获得另一种结果，即使数据库内部的数据根本没有改变过。你可能喜欢思考这种情况的隐含意思。

11.11 练习 II

11.3 当显示一张表时（例如，显示在屏幕上），表中的行是按照从上到下的顺序排列的，或者按照 ORDER BY 显示说明的顺序排列，如果省略 ORDER BY，就按照随机顺序显示（可能由系统决定，也可能是非预期的）。但是列如何按照从左到右的顺序显示呢？

11.4 按照下列 **Tutorial D** 的格式，写出相应的 SQL 格式。

a. `P { PNO } MINUS ( SP WHERE SNO = 'S2' ) { PNO }`

b. `( S { CITY } INTERSECT P { CITY } ) UNION P { CITY }`

c. `S { SNO } MINUS ( S { SNO } MINUS SP { SNO } )`

d. `SP MINUS SP`

e. `S INTERSECT S`

f. `( S WHERE CITY = 'Paris' ) UNION ( S WHERE STATUS = 20 )`

g. `S JOIN SP JOIN P`

h.
```
((S RENAME {CITY AS SC }){ SC}) JOIN
(( P RENAME {CITY AS PC }) {PC})
```

11.5 写出下列查询的 SQL 格式，即“查询供应商号码对、*X* 和 *Y*，要求供应商 *X* 和 *Y* 位于同一个城市”。

11.12　答案Ⅱ

11.3　在 SQL 中这不算是问题，因为 SQL 表的列就是按照从左到右次序排列的。从左到右的顺序显然与表中定义的顺序相同（我确信，所有的 SQL 产品都遵守这个简单的规则）。对于像 **Tutorial D** 这样的关系语言来说，这也不算是问题。因为可以通过我们正在讨论一些非关系型操作来定义。当然，真正的关系型系统必须要设定某种规则来决定结果显示的顺序。然而，这个规则对于纯粹的、系统支持的关系型语言（如 **Tutorial D** 或其他）是不可见的。

11.4　下列答案并不是唯一的。

a.
```
SELECT PNO FROM P
EXCEPT CORRESPONDING
SELECT PNO FROM SP
WHERE  SNO = 'S2'
```

b.
```
( SELECT CITY FROM S
  INTERSECT CORRESPONDING
  SELECT CITY FROM P )
UNION CORRESPONDING
SELECT CITY FROM P
```

注意该格式中的括号。当然，上面的表达式还可以简化（由读者自行完成）。

c.
```
SELECT SNO FROM S
EXCEPT CORRESPONDING
 ( SELECT SNO FROM S
   EXCEPT CORRESPONDING
   SELECT SNO FROM SP )
```

这个表达式也可以简化（由读者自行完成）。

d.
```
SELECT * FROM SP
EXCEPT CORRESPONDING
SELECT * FROM SP
```

当然，这个结果是空集。事实上，这个表达式逻辑上完全等价于如下格式：

```
SELECT * FROM SP WHERE FALSE
```

e.
```
SELECT * FROM S
INTERSECT CORRESPONDING
SELECT * FROM S
```

该表达式的结果完全等同于表 S 的当前值。事实上，该表达式逻辑上完全等价于如下格式：

```
SELECT * FROM S WHERE TRUE
```

WHERE TRUE 有时也可以省略。

f.
```
SELECT * FROM S WHERE CITY = ‘Paris’
UNION CORRESPONDING
SELECT * FROM S WHERE STATUS = 20
```

该表达式逻辑上完全等价于如下格式：

```
SELECT * FROM S WHERE CITY = 'Paris' OR STATUS = 20
```

g.
```
SELECT * FROM S NATURAL JOIN SP NATURAL JOIN P
```

h.
```
SELECT DISTINCT S.CITY AS SC , P.CITY AS PC FROM S , P
```

11.5
```
SELECT X.SNO AS XNO , Y.SNO AS YNO
FROM   S AS X , S AS Y
WHERE  X.CITY = Y.CITY
AND    X.SNO < Y.SNO
```

注意，这里使用了域变量 X 和 Y。但是，我们需要 DISTINCT 吗？

第 12 章

SQL 运算符 Ⅱ

功能在猜测中被抑制。

——William Shakespeare: *Macbeth* (1606)

第 5 章中，在 20 世纪 60 年代末和 70 年代初 Codd 定义的初始运算符基础上，我描述了一些特定的附加关系运算符，并专门讨论了（a）MATCHING 和 NOT MATCHING；（b）EXTEND；（c）快照关系；（d）聚集、归纳及其他。下面我们来看看 SQL 是如何提供这些相关功能的。

12.1 MATCHING 与 NOT MATCHING

下面先给出在第 5 章介绍半联接运算时使用的查询（在 **Tutorial D** 中为 MATCHING），即查询 1：“查询至少提供了一种零件的供应商的所有信息”。**Tutorial D** 的格式如下：

```
S MATCHING SP
```

SQL 中没有直接与 MATCHING 一致的运算符，但是通过在子查询中使用 IN 来完成查询：

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  SNO IN
     ( SELECT SNO
       FROM SP )
```

回顾第 11 章的练习 11.1 的答案，SQL 的子查询基本上是封装在括号里的 SELECT-FROM-WHERE 表达式。至于 IN 运算符，则是一个布尔表达式：

```
rx IN tx
```

如果由行表达式 *rx* 说明的行 *r* 出现在由基本表表达式 *tx* 说明的表 *t* 中，那么该表达式返回值为 TRUE，否则返回值为 FALSE。因此，在该例子中，首先返回子查询的值，然后返回如图 12.1 所示的表。

SNO
S1
S2
S3
S4

图 12.1　查询 1 的子查询结果

然后计算外查询的值，返回值表 S 的子集，即四个供应商之一。

实际上，这里讨论的查询可以采用很多种 SQL 格式来表达（实际上，每一个查询都是这样）。下面给出另外一种表达方式，使用 EXISTS 代替 IN：

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  EXISTS
     ( SELECT SNO
       FROM SP
       WHERE SP.SNO = S.SNO )
```

也可以使用自然联接来表示：

```
SELECT DISTINCT SNO , SNAME , STATUS , CITY
FROM   S NATURAL JOIN SP
```

注意后面的这个格式只是用来说明 S 和 SP 的半联接定义（参见第 5 章）。一句话的忠告：这里 SELECT 子句中若用“S.*”代替列的逗号列表，则会产生错误。在 *SQL and Relational Theory* 中解释了复杂的原因。而且也不能采用“*”代替列的逗号列表。

现在转到半差运算（等同于 **Tutorial D** 中的 NOT MATCHING），仍然采用第 5 章中的查询来介绍这个运算符，即“查询没有供应零件的供应商信息”。**Tutorial D** 的格式如下：

```
S NOT MATCHING SP
```

SQL 中没有与之对应的格式，但是可以采用 NOT IN 子查询的方式来实现，如下：

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  SNO NOT IN
     ( SELECT SNO
       FROM SP )
```

当然，表达式 *rx* NOT IN *tx* 逻辑上等价于 NOT (*rx* IN *tx*)。

另外，也可以采用 NOT EXISTS 实现上述查询：

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  NOT EXISTS
     ( SELECT SNO
       FROM SP
       WHERE SP.SNO = S.SNO )
```

并没有明显的方法在该查询中使用 NATURAL JOIN，除非我们只是想按顺序阐明 S 和 SP 之间的半差的定义（见第 5 章），具体如下：

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
EXCEPT CORRESPONDING
SELECT SNO , SNAME , STATUS , CITY
FROM   S NATURAL JOIN SP
```

12.2 EXTEND

这里仍然使用第 5 章的查询来介绍 EXTEND，即“查询每个零件的详细信息，用克来表示零件的重量”。**Tutorial D** 的格式如下：

```
EXTEND P : { GMWT := WEIGHT * 454 }
```

SQL 的格式如下：

```
SELECT PNO , PNAME , COLOR , WEIGHT , CITY ,
( WEIGHT * 454 ) AS GMWT
FROM   P
```

注意：SQL SELECT 子句确实涉及很多领域，在第 11 章中我们已经看到与关系投影和改名操作对应的 SQL 格式都是依据该子句实现的。现在我们来看看如何依赖该子句实现 EXTEND，在本章后面我们还将看到该子句的另一个作用。

第 5 章中关于 EXTEND 的第 2 个例子如下：

```
( ( EXTEND P : { GMWT := WEIGHT * 454 } )
                         WHERE GMWT > 7000.0 )
                                       { PNO , GMWT }
```

这个查询的目的是获得零件重量大于 7000 克的所有零件号码及重量（用克表示）。下面采用 SQL 格式来实现该查询：

```
SELECT PNO , ( WEIGHT * 454 ) AS GMWT
FROM   P
WHERE  GMWT > 7000.0
```

然而，我希望您已经发现这个格式是不正确的。原因是 GMWT 是结果表中的一个列名，而表 P 中没有此列名，因而 WHERE 子句没有任何实际意义，在编译时该表达式就会报错。注意：问题的原因是我们需要在限制条件之前进行扩展（就像

Tutorial D 的格式），但 SQL 却是在扩展之前执行限制（回顾第 11 章中给出的 SELECT-FROM-WHERE 表达式求值的概念性算法，已经表明轴线判断 FROM 子句，然后是 WHERE 子句，最后是 SELECT 子句）。

现在，修正前面问题的一种方法就是在 WHERE 子句中用定义的表达式来代替对 GMWT 的引用，即：

```
SELECT PNO , ( WEIGHT * 454 ) AS GMWT
FROM   P
WHERE  ( WEIGHT * 454 ) > 7000.0
```

当然，子表达式 WEIGHT * 454 在整个表达式中出现了 2 次，我们希望可以有 1 个优化器只识别计算该子表达式 1 次，而不是 2 次。

因此，另外一种方式就是采用与 **Tutorial D** 有些近似的方式实现该查询（为了清晰地表示该方式，采用完全显示的域运算符来显示说明）：

```
SELECT TEMP.PNO , TEMP.GMWT
FROM ( SELECT PNO , ( WEIGHT * 454 ) AS GMWT
       FROM P ) AS TEMP
WHERE TEMP.GMWT > 7000.0
```

注意该例子中FROM子句的子查询，也注意名字TEMP指的是子查询计算后得到的结果表[1]。

下面多列举一些第 5 章给出的 **Tutorial D** 格式表示的 EXTEND 例子，采用 SQL 格式表示的语法：

1. `EXTEND S : { TAG := 'Supplier' }`

   SQL 格式如下：

```
SELECT SNO , SNAME , STATUS , CITY , 'Supplier' AS TAG
FROM   S
```

2. `EXTEND ( P JOIN SP ) : { SHIPWT := WEIGHT * QTY }`

   SQL 格式如下：

```
SELECT PNO , PNAME , COLOR , WEIGHT , CITY , SNO , QTY ,
                                     ( WEIGHT * QTY ) AS SHIPWT
FROM   P NATURAL JOIN SP
```

3.
```
( ( EXTEND P : { GMWT := WEIGHT * 454 } ) { ALL BUT WEIGHT } )
                                          RENAME { GMWT AS WEIGHT }
```

注意，这个表达式的含义是“如果…，那么…”的查询，SQL 格式如下：

---

1 实际上，TEMP 不是一个真正的表的名字，相反，它是一个域变量，就像在第 11 章中定义的一样。而且要注意 SQL 规则规定了 FROM 子句的子查询必须与用 AS 显式定义的域变量相一致，即使整个表达式中没有显式引用这个域变量。

```
SELECT PNO , PNAME , COLOR , ( WEIGHT * 454 ) AS WEIGHT , CITY
FROM   P
```

12.3 映像关系

SQL 没有直接等价于 **Tutorial D** 格式的映像关系，因此，在 **Tutorial D** 中采用映像关系表示的查询很难用 SQL 表示。这里仅仅给出一个例子，回顾下面的 **Tutorial D** 表达式，即“查询供应了所有零件的供应商信息”。**Tutorial D** 格式如下：

```
S WHERE ( ‼SP ) { PNO } = P { PNO }
```

一种可能的 SQL 格式如下：

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P
       WHERE  NOT EXISTS
            ( SELECT *
              FROM   SP
              WHERE  S.SNO = SP.SNO
              AND    SP.PNO = P.PNO ) ) ;
```

该查询的含义是：获得供应商 *s* 的信息，对于该供应商 *s*，不存在零件 *p*，它提供了 *p* 但不存在于 *SP* 中。或者理解为：获得这样供应商的信息，即不存在一个零件，它没有供应（注意采用了双重否定）。

12.4 聚集和归纳

如果你需要重新理解聚集和归纳的概念，请回顾第 5 章，该章中提供了很多的资料可以帮你复习，当然只需要关注那些与 SQL 一致的特征，即：

- 典型的聚集运算符，如 COUNT、SUM、AVG、MAX、MIN、AND、OR 及 XOR，这些运算符的作用是从关系的某些属性值的聚集（如集合或包）来产生一个结果值（COUNT 有些特殊，聚集时作用于整个关系）；
- 聚集运算符有 2 个作用，一是用于归纳，二是用于第 5 章提到的“衍生的限制”。回顾一下，映像关系几乎两种情况都会涉及。

我们已经知道 SQL 不支持映像关系，现在我要声明它也不支持聚集运算符。（如果你恰好对 SQL 有所了解的话，后面的声明会让你感到吃惊。）但是它确实不支持归纳和衍生的限制。事实上，它有 2 个特征（即 GROUP BY 和 HAVING）可以用来显式说明处理这样的事情，下面将详细讨论。

12.4.1　归纳

首先，采用 **Tutorial D** 格式来表示如下的查询，即查询 1："对于每个供应商，查询供应商号码及提供的零件数量"。

```
EXTEND S { SNO } : { PCT := COUNT ( ‼ SP ) }
```

查询结果如图 12.2 所示。

SNO	PCT
S1	6
S2	2
S3	1
S4	3
S5	0

图 12.2　查询 1 的结果列表

采用 SQL 表示格式如下：

```
SELECT S.SNO , ( SELECT COUNT ( PNO )
                 FROM SP
                 WHERE SP.SNO = S.SNO ) AS PCT
FROM   S
```

注意该例中 SELECT 子句中的子查询（采用 AS 规则说明与结果列的列名）。就像第 11 章中提供的例子 11.1d 一样（这里首先遇到了 SQL 子查询的概念），这里需要给出一些解释。首先，该例的子查询是相关子查询（参见第 11 章例子 11.1e 的答案），意思是它包含了对外部表达式定义的某些事情的引用，更特殊的是针对每一个外部引用都需要反复进行判断。（该例子中的外部引用就是对出现在子查询 WHERE 子句中的 S.SNO 的引用。）所以你可以想象一下，对该表达式的求值要通过检查外部表（如例子中的表 S）的多行来完成，一次要检查一行。我们来考虑一下表 S 中的某行，如供应商 S1 的行，子查询就转化为如下形式：

6

图 12.3　子查询的结果表

```
( SELECT COUNT ( PNO )
  FROM SP
  WHERE SP.SNO = 'S1' )
```

该表达式将返回具有 1 行 1 列的表，如图 12.3 所示（列没有列名）[1]。

---

1 同样，该图中缺少下划线并不是错误。

现在发生了双重强制类型转换（例如，隐式的类型转换）。这个表格被迫变成一行，反过来该行只有一个标量值。这个标量值变成了整个结果行中供应商 S1 的 PCT 值（由于规则中说明为 PCT），对于表 S 中的每一行都要重复这个过程，从而得到最终的结果值。

语法注释：现在我们已经看到了 SELECT 子句的子查询、FROM 子句的子查询、WHERE 子句的子查询，但是你注意到这些子查询何时返回标量值、何时返回表吗？（事实上，它们有时既不返回标量、也不返回表格，而是返回行，但我不打算给出更详细的例子，也不给出更多的讨论。）你如何看待 SQL 规则可能看起来像在特定的环境下决定子查询的类型以及对子查询的解释？

现在应该来解释一下，前面那个查询的 SQL 格式也许不是大多数 SQL 从业人员喜欢使用的查询的方式，相反，他们更喜欢使用如下的格式：

```
SELECT SNO , COUNT ( PNO ) AS PCT
FROM   SP
GROUP  BY SNO
```

解释：假设 *t* 就是由 FROM 和 WHERE 子句产生的结果表（在该例中，*t* 只是 SP，因为没有 WHERE 子句，省略的部分就等价于 WHERE TRUE），然后计算 GROUP 子句，它也会产生一定的影响（至少从概念上理解是这样），即把表中的行分成若干组，这样的每个组都有一个唯一的值，被称作“分组的列”（在该例中，每个组都按照唯一的 SNO 进行分组）。最后，计算 SELECT 列，它重新构造每一行，每一组的每一行都由 SNO 和相应的 PCT 值组成（例如，对应的统计值）。但这其实是一个陷阱，表 SP 的供应商 S5 没有对应的行，所以在结果的分组操作中就没有供应商 S5 的分组，因此，最后的结果中就没有供应商 S5 对应的行，换句话说，其结果值如图 12.4 所示。

SNO	PCT
S1	6
S2	2
S3	1
S4	3

图 12.4 GROUP BY 查询执行结果

事实上，作为 SQL（或者 SEQUEL）的一部分，GROUP BY 从一开始就要正确处理，显式说明来帮助完成类似于上面讨论的那种查询，即（a）或者从来不使用它，或者不使用上面讨论的那种格式；（b）使用时需要小心处理，以避免产生刚刚说明的类似问题。

这个例子其实还没有介绍完，下面还要说明一点。再考虑一下如下的格式：

```
SELECT S.SNO , ( SELECT COUNT ( PNO )
                 FROM SP
```

```
                    WHERE SP.SNO = S.SNO ) AS PCT
FROM   S
```

现在看看子查询中的“COUNT（PNO）”，该部分没有构成对聚集运算符的调用！（这是因为以前我曾经声明过：SQL 对聚集运算符没有恰当的支持语法。）因此如果这个构造确实代表了一个聚集运算符的调用，我们应该可以写出如下的 SQL 赋值语句：

```
SET PCT = COUNT ( PNO ) ;
```

但很显然，我们不能。

现在考虑另一个例子（对前面例子稍微修改了一下），即查询 2：“对于每个供应商，获得其供应商号码，及供应货物的总量”。**Tutorial D** 格式如下：

```
EXTEND S { SNO } : { TOTQ := SUM ( ‼SP , QTY ) }
```

查询 2 的运行结果如图 12.5 所示。

SNO	TOTQ
S1	1300
S2	700
S3	200
S4	900
S5	0

图 12.5　查询 2 的运行结果

SQL 格式如下：

```
SELECT S.SNO , ( SELECT SUM ( QTY )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

但是现在又产生了另一个陷阱：供应商S5 的供应总量值是一个空集的总和。从逻辑上来说，空集的总和当然是 0，但是SQL把它定义为空（*null*）[1]，因此在该例中产生了如下不正确的结果，如图 12.6 所示。

SNO	TOTQ
S1	1300
S2	700
S3	200
S4	900
S5	

图 12.6　查询 2 的不正确运行结果

1 事实上，在 SQL 中，如果变量是空值的话，SUM、AVG、MAX、MIN、EVERY、SOME 运算也都会返回空值(最后两个运算与 Tutorial D 中的 AND 和 OR 运算特别相似)。(SQL 中没有 XOR 运算）COUNT 是特殊的一个，如果它的变量为空，则返回结果为 0。

该例子只是说明了许多特殊情况中的一种，即 SQL 表达式可以返回带有空值的表，即使输入的值不包含任何的空值。因此我们必须要避免空值的产生，我们所需要的是有一个运算符能够在空值出现的时候用一个真实的值替代它（在它还没有机会产生破坏之前），这个运算符叫作 COALESCE。因此，前面那个 SQL 表达式修正如下：

```
SELECT S.SNO , ( SELECT COALESCE ( SUM ( QTY ) , 0 )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

解释如下：通常情况下，表达式COALESCE (*exp1*,*exp2*,...,*expn*)返回表达式列表（按照从左到右的顺序）中第一个表达式的值，来代表一个真实的值，从而替代空值，这里*exp1*,*exp2*,...,*expn*是一个具有相同类型的、采用逗号分隔的表达式列表[1]。因此在该例中，如果给定的供应商至少存在一种供应关系，那么就会得到正确的总和。但如果某些供应商根本不存在供应关系，则SQL就会返回空值，那么COALESCE就立即用 0 替代空值。

12.4.2 “通用的限制”

在第 5 章我使用术语“通用的限制”（*generalized restriction*）来表示 *r* WHERE *bx*，这里布尔表达式 *bx* 需要调用聚集运算符，而且也会引用一个映像关系（至少在 **Tutorial D** 中是这样的），下面给出一个例子（查询 3：查询供应关系小于 3 的供应商信息）：

```
S WHERE COUNT ( ‼SP ) < 3
```

查询 3 的运行结果如图 12.7 所示。

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

图 12.7　查询 3 的运行结果

SQL 的格式如下（请注意 WHERE 子句中的子查询）：

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE ( SELECT COUNT ( PNO )
        FROM   SP
        WHERE  SP.SNO = S.SNO ) < 3
```

1 当然，最好存在这样的表达式，因为 COALESCE 调用本身就会返回空值。

然而，这种格式不是大多数习惯使用 SQL 的用户所希望使用的格式。相反，大多数习惯于 SQL 格式的用户会使用如下格式：

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S NATURAL JOIN SP
GROUP  BY SNO
HAVING COUNT ( PNO ) < 3
```

解释如下：HAVING 子句相当于每个分组后的 WHERE 子句，也就是说，进一步讨论后会消除某些特定的组，就像使用 WHERE 子句会消除某些行一样。（注意：SQL 表达式中，HAVING 子句也可以使用 GROUP BY。）但这又是一个陷阱，供应商 S2 和 S3 的分组都通过了 HAVING 的检测，所以这些供应商的信息会出现在最后的结果中（相反，供应商 S1 和 S4 没有通过检测）。但对于供应商 S5，却没有分组，这是因为在 NATURAL JOIN SP 运算后的结果里不存在供应商 S5 的行，所以最后的结果中就没有显示供应商 S5！换句话说，其运行结果如图 12.8 所示。

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris

图 12.8 查询 3 的 SQL 运行结果

事实上，显式说明 HAVING 可以有助于完成上述查询，（a）要么不使用它；（b）如果使用，需要注意它的正确用法（就像 GROUP BY）。

下面再举一个例子，查询 4：“查询供应总量少于 1000 的供应商信息”。**Tutorial D** 的格式如下：

```
S WHERE SUM ( ‼SP , QTY ) < 1000
```

SQL 格式如下（注意 COALESCE 的用法，其目的是保证最后的结果中存在供应商 S5 的信息）：

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE ( SELECT COALESCE ( SUM ( QTY ) , 0 )
        FROM   SP
        WHERE  SP.SNO = S.SNO ) < 1000
```

最后再给出一个稍复杂一些的例子，查询 5：“查询供应总量达于 250 的供应商号码以及供应数量”。**Tutorial D** 的格式如下：

```
( EXTEND S { SNO } : { TOTQ := SUM ( ‼SP , QTY ) ) WHERE TOTQ > 250
```

SQL 格式如下：

```
SELECT TEMP.SNO , TEMP.TOTQ
FROM ( SELECT SNO , ( SELECT COALESCE ( SUM ( QTY ) , 0 )
                      FROM SP
                      WHERE SP.SNO = S.SNO ) AS TOTQ
       FROM   S ) AS TEMP
WHERE  TEMP.TOTQ > 250
```

12.5 练习

给出下列 **Tutorial D** 表达式的 SQL 语法格式：

a. `P MATCHING S`

b. `S NOT MATCHING ( SP WHERE PNO = 'P2' )`

c. `EXTEND P : { SCT := COUNT ( ‼SP ) }`

d. `P WHERE SUM ( ‼SP , QTY ) < 500`

12.6 答案

a.
```
SELECT PNO , PNAME , COLOR , WEIGHT , CITY
FROM   P
WHERE  CITY IN ( SELECT CITY
                 FROM   S )
```

b.
```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  SNO NOT IN ( SELECT SNO
                    FROM   SP
                    WHERE  PNO = 'P2' )
```

c.
```
SELECT PNO , PNAME , COLOR , WEIGHT , CITY ,
                                      ( SELECT COUNT ( SNO )
                                        FROM SP
                                        WHERE SP.PNO = P.PNO ) AS SCT
FROM   P
```

d.
```
SELECT PNO , PNAME , COLOR , WEIGHT , CITY
FROM   P
WHERE  ( SELECT COALESCE ( SUM ( QTY ) , 0 )
         FROM   SP
         WHERE  SP.PNO = P.PNO ) < 500
```

说明：答案不是唯一的，以上答案仅供参考。

第 13 章

SQL 约束

黄金法则可以保证数据库处于最佳状态。

——Anon: *Where Bugs Go*

回顾第 6 章讲过的完整性约束（简称约束），不严格地讲，约束是一个布尔表达式，结果必须为 TRUE（否则，数据库中就会产生错误）。再回顾一个黄金法则（**The Golden Rule**），即要求完整性约束必须要满足一组声明范围。换句话说，一个单独的声明就是一个完整性单元，没有声明（特别是，没有修改声明）就会使数据库处于不一致的状态。在本章中，我们来仔细看看 SQL 的相关特征。

13.1 数据库约束

在 SQL 中数据库约束采用 CREATE ASSERTION 定义，与 **Tutorial D** 的 CONSTRAINT 声明是一致的。在第 6 章中讨论了 5 个可能的“商业规则”，并通过实例展示了如何使用 CONSTRAINT 声明来实现这些规则。现在我们来看看与 CONSTRAINT 声明相似的 CREATE ASSERTION。注意，为了方便比较，还是先给出 **Tutorial D** 格式。

1．供应商的状态值必须在 1 至 100 之间。

Tutorial D 格式：

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

SQL 格式：

```
CREATE ASSERTION CX1
       CHECK ( NOT EXISTS ( SELECT * FROM S
                            WHERE STATUS < 1 OR STATUS > 100 ) ) ;
```

通过例子可以看出，CREATE ASSERTION 声明包含以下几个部分：（a）关键

词 CREATE ASSERTION；（b）约束的名字，要求紧跟着 CREATE ASSERTION；（c）关键词 CHECK，要求紧跟着约束名；（c）放在括号中的布尔表达式（必须为 TRUE），要求紧跟着 CHECK。

你可能想到了 **Tutorial D** 的另一种格式：

```
CONSTRAINT CX1 AND ( S , STATUS ≥ 1 AND STATUS ≤ 100 ) ;
```

SQL的另一种格式为[1]：

```
CREATE ASSERTION CX1
       CHECK ( ( SELECT COALESCE ( EVERY ( STATUS >= 1 AND
                                           STATUS <= 100 ) , TRUE )
                  FROM S ) ) ;
```

2．伦敦供应商的状态值必须为 20。

Tutorial D 格式：

```
CONSTRAINT CX2 IS_EMPTY ( S WHERE CITY = 'London' AND STATUS ≠ 20 ) ;
```

SQL 格式：

```
CREATE ASSERTION CX2
       CHECK ( NOT EXISTS ( SELECT * FROM S
                            WHERE CITY = 'London' AND STATUS <> 20 ) ) ;
```

3．供应商号码必须唯一。

Tutorial D 格式：

```
CONSTRAINT CX3 COUNT ( S ) = COUNT ( S { SNO } ) ;
```

SQL 格式：

```
CREATE ASSERTION CX3
       CHECK ( ( SELECT COUNT ( SNO ) FROM S ) =
               ( SELECT COUNT ( DISTINCT SNO ) FROM S ) ) ;
```

针对该例强调几点。

- 首先，注意最后一行的 COUNT（DISTINCT SNO）。在本书中我们第一次看到这样一种结构。不严格地讲，该结构的含义是将 COUNT 运算符对 S 中 SNO 进行投影运算（即真正的关系投影，这样可以消除重复）。为了对比差异，考虑 COUNT（STATUS）和 COUNT（DISTINCT STATUS），仍然用我们常用的例子做样本数据，前者的返回值为 5，后者的返回值为 3。
- 第二，前面 SQ 格式的有效性不能立刻显示出来。实际上有时会发生一些

1 这里给出的 COALESCE 实际上不是必须的，但放在这里也没有错。详细讨论请参见 *SQL and Relational Theory*。

神秘的事情。如你所见，CHECK 声明的布尔表达式会涉及等值比较，要求操作数必须用查询来表示，而子查询的结果是表格，因此就要比较两个表格的等价性（然而在第 11 章中我们看到 SQL 不支持这样的比较）。所以如何进行比较呢？我不想在这里揭开这个谜底，把它留给读者去考虑。

- 第三，还有一种更简单的方法来声明 SQL 约束，即：

```
CREATE ASSERTION CX3
       CHECK ( UNIQUE ( SELECT SNO FROM S ) ) ;
```

当且仅当变量表中的行都不相同时，SQL 的 UNIQUE 运算符返回值才为 TRUE。因此，当且仅当表 S 中不存在供应商号相同的两行时，UNIQUE 结果才返回 TRUE。

- 当然，实际上我们可以不通过显式CREATE ASSERTION声明来表示约束，而是通过在表S的定义中加入UNIQUE (SNO)的形式来表示 [1]。但有趣的是，这种说明实际上只是表达形式上的简化，显然显式的CREATE ASSERTION声明有点冗长。

4．供应状态小于 20 的供应商不能供应零件 P6。

Tutorial D 格式：

```
CONSTRAINT CX4
    IS_EMPTY ( ( S JOIN SP ) WHERE STATUS < 20 AND PNO = 'P6' ) ;
```

SQL 格式：

```
CREATE ASSERTION CX4
       CHECK ( NOT EXISTS ( SELECT * FROM S NATURAL JOIN SP
                            WHERE STATUS < 20 AND PNO = 'P6' ) ) ;
```

这是在列表中涉及两个基本表的第一个例子（即基本表 S 和 SP）。当然，通常情况下约束与基本表的数量是无关的。而且更重要的一点是，SQL 需要在事务提交时检测相应的约束，因而，更不幸的是 SQL 有时不能支持黄金法则（**The Golden Rule**）。

5．表 SP 的每个供应商号码必须出现在表 S 中。

Tutorial D 格式：

```
CONSTRAINT CX5 SP { SNO } ⊆ S { SNO } ;
```

---

1 仅仅是为了兴趣，现在我来告诉你，在表 S 中使用 UNIQUE（SNO）说明的标准语法格式（我采用标准通用的语法来表示这个特殊的例子）：当且仅当 EXISTS(SELECT * FROM S WHERE NOT(UNIQUE(SELECT SNO FROM S)))为真时，UNIQUE（SNO）是不满足条件的。我希望已经非常清楚地解释了这个例子。

SQL 格式：

```
CREATE ASSERTION CX5
       CHECK ( NOT EXISTS ( SELECT SNO FROM SP
                            EXCEPT CORRESPONDING
                            SELECT SNO FROM S ) ;
```

这个例子也涉及两个基本表。当然我们也可以不采用显式 CREATE ASSERTION 声明的形式，而是通过在表 SP 定义中加入 FOREIGN KEY 说明来表示这个约束。但是，和上面第 3 个例子一样，这种说明实际上只是表达形式上的简化，显然显式的 CREATE ASSERTION 声明有点冗长。

对基本表和列约束的说明：前面的例子足以说明，在 **Tutorial D** 中采用 CONSTRAINT声明的任何约束，都可以采用SQL的CREATE ASSERTION进行声明[1]。但是SQL有一个特点，即这样的约束都可以在基本表的定义中作为定义的一部分出现来替代单独的声明，例如，基本表约束（*base table constraint*）。（**Tutorial D** 只在定义中支持码和外码的约束，而不支持其他的定义。）而且，SQL还有一个特点，即一些约束可以通过列约束（*column constraint*）的形式来声明（这种声明不仅是基本表定义的一部分，而且是基本表中特定列定义的一部分），例如，只涉及一列的NOT NULL约束和码约束。但是这些特征基本上只是语法上的一些装饰，进一步的讨论已经超出了本书的范围。

13.2 类型约束

在前面第 10 章我已经多次提到，SQL 确实允许用户定义自己需要的类型。现在这已经成为了一条公理（至少有人是这样认为的），即某些用户自定义类型 *T* 定义中的部分必须确保是可以构成类型 *T* 的值的规格说明。通过例子具体说明一下，假设我们要把表 SP 的数量列（QTY）的类型采用用户自定义类型，而不是采用系统定义的类型 INTEGER（本书中一直假设使用 INTEGER 类型）。为了简化问题，假设表 SP 中该列的类型名为 QTY，与列名一致，那么我们要确定一种表示合法数量的方式（即类型 QTY 的唯一合法值），即确保在整数 1 至 5000 之间取值。在 **Tutorial D** 中，采用下面的大致格式定义 QTY 类型：

```
TYPE QTY ... CONSTRAINT Q > 1 AND Q < 5000 ... ;
```

---

1 除非假设 SQL 约束中不包含“可能未确定的表达式”（参见第 11 章关于这个术语的解释）。如果真是这样的话，在实际情况中可能会引起严重的问题，特别是，如果表格表达式包含了一些字符串类型的子表达式（这是一种很常见的情况），则几乎所有的表格表达式都会被看作是“可能没有确定值的”。详细讨论请参见 *SQL and Relational Theory*。

符号*Q*表示该类型的任一值（实际上，是任一数量值）[1]。

上面的 **Tutorial D** 格式中使用了关键字 CONSTRAINT，这也是合理的，因为在该例中就是要保证 QTY 的值位于一个合理的范围之内。换句话说，我们定义了一个类型约束（*type constraint*），即构成特定类型的合法值的规格说明。注意，在第 6 章讨论的约束（在本章前面的章节中也讨论过）被称为数据库约束（*database constraint*），目的是为了和类型约束加以区分。实际上，术语约束（*constraint*）的使用是不加限制的，但通常用来表示数据库约束，除非在上下文环境中加以特殊说明。

在第 6 章中没有提到类型约束，这是因为：第一，它们本身并不是关系理论的一部分（相反，却是相关联的类型理论的一部分）；第二，我或多或少想当然地认为，允许用户自定义类型的系统也必须允许用户自己定义构成这些类型的值。但是我错了，SQL 不允许这样。

我来解释一下原因：SQL 允许我们把数量的类型定义为 QTY，它也允许采用整型来表示该类型的值，但是它不允许把该类型的合法值限定在 1 至 5000 之间。因此，在 SQL 中数量 1,000,000,000 显然是合法的！（所以，−1,000,000,000 也是合法的。）

当然，我们可以通过 CREATE ASSERTION 来弥补 SQL 中不支持类型约束的不足（以相当大的代价）。假设 *Q* 代表类型 QTY，*T* 代表包含该类型的基本表，则可以表示为如下格式：

```
CREATE ASSERTION CXQ
       CHECK ( NOT EXISTS ( SELECT * FROM T
                            WHERE  Q < QTY ( 1 )
                            OR     Q > QTY ( 5000 ) ) ) ;
```

表达式 QTY（1）和 QTY（5000）是 QTY 的标识符。显然，每一个这样的列 *Q* 和表 *T* 都可以采用 CREATE ASSERTION 进行说明。

13.3 练习

13.1 按照下面的“商业规则”，写出供应商-零件数据库的 CREATE ASSERTION 声明。

a．所有红色零件的重量必须小于 50 磅。

b．不存在两个供应商位于同一个城市。

c．在同一时间至多只能有 1 个供应商所在城市为雅典。

1 这个解释有些过于简单，但就目前情况来说，这就足够了。

d．至少存在一个伦敦的供应商。

e．供应商 S1 和零件 P1 必须位于不同的城市。

13.2 你钟爱的 SQL DBMS 支持 CREATE ASSERTION 吗？

13.3 如果问题 13.2 的答案是“no”，请说明它支持其他什么类型的约束？

13.4 你如何处理不能进行声明的那些约束？

13.5 请问 SQL 中为什么在检测约束条件（如约束 CX4）时要进行推迟检测？

13.4 答案

13.1 a.
```
CREATE ASSERTION CXA
       CHECK ( NOT EXISTS ( SELECT * FROM P
                            WHERE COLOR = 'Red' AND WEIGHT >= 50 ) ) ;
```
b.
```
CREATE ASSERTION CXB
       CHECK ( ( SELECT COUNT ( SNO ) FROM S ) =
               ( SELECT COUNT ( CITY ) FROM S ) ) ;
```
c.
```
CREATE ASSERTION CXC
       CHECK ( ( SELECT COUNT ( SNO ) FROM S
                 WHERE CITY = 'Athens' ) < 2 ) ;
```
d.
```
CREATE ASSERTION CXD
       CHECK ( EXISTS ( SELECT * FROM S
                        WHERE CITY = 'London' ) ) ;
```
e.
```
CREATE ASSERTION CXE
       CHECK ( ( SELECT COUNT ( CITY )
                 FROM ( SELECT CITY FROM S
                        WHERE SNO = 'S1'
                        UNION CORRESPONDING
                        SELECT CITY FROM P
                        WHERE PNO = 'P1' ) AS POINTLESS ) < 2 ) ;
```

13.2 答案也许为“no”。实际上，完整性约束已经构成了一个特定的领域，该领域中的标准要比 DBMS 的大部分标准（不是所有的产品）好的多，即使目前还不是很完美。

13.3 大部分产品都支持 UNIQUE 约束、PRIMARY KEY 约束（在 UNIQUE 约束的语法上稍微有一些变化）、FOREIGN KEY 约束以及其他一些简单的“基本表”约束。后面的基本表约束通常可以通过单独检测每一行来实现约束条件的检测。换句话说，约束条件中的布尔表达式被限制为基本表级别的限制条件。（在 SQL 的术语中，约束条件不允许包含子查询。）

13.4 这些人可能会在某些地方来加入一些恰当的应用程序代码（可能是“触发器”的形式）。详细讨论请参见 *SQL and Relational Theory*。

13.5 实际上，原因就是 SQL 不支持类似于 **Tutorial D** 的多变量形式（参见第 6 章），多变量可以允许用户在一个语句中修改两个或多个表。例如，约束 CX4，可以执行下面的 UPDATE：

```
UPDATE S
SET    STATUS = 10
WHERE  SNO = 'S1' ;
```

如果此时检查约束条件，UPDATE 操作必然失败。但是如果把检测推迟到下面的 DELETE 操作之后：

```
DELETE
FROM   SP
WHERE  SNO = 'S1'
AND    PNO = 'P6' ;
```

那么，这个修改就成功了。然而，如果采用这种方式来推迟检测，那么在从 UPDATE 到 DELETE 操作之间数据库就会处于不一致状态，因此 SQL 就破坏了**黄金法则**（**The Golden Rule**）。特别是如果在此期间，执行修改操作的程序发出询问："存在状态值小于 20 的、供应了零件 P6 的供应商吗？" 答案一定是 ***yes***。

注意：奇怪的是 SQL 实际上支持多变量格式！例如，下面的 SQL 语句完成了把变量 X、Y、Z 的值分别赋值给变量 A、B、C。

```
SET ( A , B , C ) = ( X , Y , Z ) ;
```

（实际上，从技术实现角度来说，这是一个行赋值语句。）但麻烦的是，赋值语句中的目标变量不允许赋值给表变量 [1]。使得事情变得更加奇怪的是，SQL确实能够执行一些特定的多个表变量赋值，至少采用隐式的方式，例如，当它执行"级联删除时"（参见第 3 章练习 3.4 的答案）。

1 假定 SQL 中根本没有提供恰当的表变量定义的前提下，你也许不会感到这样惊讶。

第 14 章

SQL 与关系模型

在理想与现实之间，往往会有阴影降临。

——T.S.Eliot：*The Hollow Men*（1925）

从前面几章的介绍可以看出，SQL 与关系模型不是一回事，或者更具体一点地说，SQL 被看作是一种具体的关系型语言，但显然 SQL 在很多方面都无法把想法和潜在的抽象关系模型相对应。实际上，SQL 要忍受两方面的过错，即省略的过错和代理的过错，一方面它不能正确支持（或者根本就不支持）关系模型的很多地方，另一方面它所支持的地方又不能与任何关系模型相对应。（当然，在整本书中我只把注意力局限在这里，即只关心 SQL 的核心特征。）省略的一些例子可以很好地支持等价、表类型、关系代数运算符。非关系型特征的例子有空值、重复行、从左到右排序的列等。

本书中我的主要目的是描述和解释关系模型，而不是 SQL。但是我认为给出 SQL 的普遍存在的特征（更确切地说，不论好坏，有一点可以坚持，即数据库专业人员确实需要面对它，利用它处理实际问题）可以为专业人士提供多种方法解决问题，SQL 的这些方法都违背了抽象模型的一些规定。实际上，我非常相信这样的专业人士对关系模型本身已经了如指掌，知晓模型和 SQL 之间的差异可以对他们有所帮助，因此本章内容就来讲解 SQL 与关系模型之间的差异。

14.1 概述

再重复一下我在第 10 章讲过的内容，我确信先了解关系模型，然后再学习SQL会比先学习SQL再学习关系模型要容易些。其原因是如果先学习SQL再学习关系模型的话，会需要很多未了解的知识（因此本书才按照这样的结构来安排，就像在第 1 章中介绍的一样）。实际上，我相信并不是任何人都真正了解SQL的方方面面，

只是知道了这种语言，而不是真正了解它的实质。SQL如此庞大、如此复杂、如此特别、如此非正交（此处忽略了它的逻辑差异、不一致性、矛盾等[1]），经过分析后我不得不相信SQL是很难教会的。我不止一次地遇到过这样的情况，即我向很多SQL专家请教一些技术问题的答案时（我曾拜访过SQL标准委员会的成员或曾经的成员），往往要等上一些时间才能获得答案，即使答案是现成的（但不总是这样的），这些答案也不保证一直是正确的。

无论如何，我希望你从上面的叙述中可以了解到，为什么人们（就像我自己）都拒绝把现在主流的“关系型”产品看作是关系型的。实际上它们是 *SQL* 型产品：它们支持的基本数据对象是 SQL 表，不是关系。它们支持的运算符可以处理 SQL 表，而不能处理关系。但令人遗憾的是（不能说是奇怪的），我一直坚信真正的关系型 DBMS 应该远远超越只支持 SQL 的 DBMS（不仅是从可用性的角度超越，而且要在易实现性、优化性和良好的性能方面来超越）。我简短地阐述一下。

- **可用性**：一个不可否认的事实就是，SQL 要比关系模型复杂得多，而且它也没额外提供一些有用的功能。实际上严格来讲，它提供的功能要比关系模型少，因为它完全不是关系型的（参见第 7 章中关于此概念的讲解）。
- **优化和性能**：首先我需要解释一下优化器的概念，优化器的功能之一就是完成表达式的转换，即将表达用户原始请求的表达式转换为另一种逻辑上与其等价的表达式，并且要保证得到等价的结果，而且性能要优于原始的表达式（至少这是我们所希望的）。（我已经在本书中的某些地方提到过这个概念，比如第 1 章。）也就是说，对于关系来说可以完成很多这样的转换，但是对于带有重复行的表来说不能完成这么多的工作，如果考虑列的顺序、考虑空值则不能完成众多的转换。换句话说，这些非关系型的特征（重复行、列序、空值）都是阻碍优化的重要因素，因此会影响性能，另一方面，这些特征也会使优化器更加复杂，所以很难实现。

注意：这里我又提出了几点，首先，你需要理解“不存在两个等价的 SQL”（我在第 10 章曾提到过）。

- 目前没有一款商业产品完全支持这个标准（实际上，如果考虑到不一致性和我上面提到的那些，没有一款产品能做到这一点）。
- 同时，每款产品都具有自身特有的一些特征，而这些特征并不是标准的一部分。
- 而且，这个标准明显遗留了一些需要在特定产品中解决的问题。（例如，

---

1 支持这些声明的证据（关于不一致性及其他特征）可以参见附录 D：SQL 标准指南（1997 年，由 Addison-Wesley 出版，第 4 版），作者：我和 Hugh Darwen。

为结果中的列命名，否则这个列就没有名字。可以参照第 10 章练习 10.3 的答案。）你认为这些产品都能恰好采用相同的方法来解决这些问题吗？

我感觉还有义务提一下，即使从纯粹的正规编程语言角度来说，以任何标准衡量SQL的设计都是很糟糕的。实际上对于语言设计的好坏已经建立起评价标准，我记得应该是：通用性、简洁性、完整性、相似性、可扩展性、开放性和正交性[1]，但似乎没有任何证据可以证明SQL的设计符合上述特性。

但有趣的是，为什么市场上喜欢使用 SQL，而不使用关系模型呢（对于此事我有自己的观点，但我不会在这里表明）。迄今为止，无论如何我们都还不得不学会忍受这种选择的结果。即便这样，对于 SQL 表的设计和 SQL 运算符的使用等规则的研究仍然是人们感兴趣的，当然目的是使整个系统看上去与真正的关系型一样。实际上我在第 11 章曾经提到过，这些规则在 *SQL and Relational Theory* 一书中用很大篇幅进行了介绍。然而不幸的是，由于 SQL 语言和当前 SQL 产品的设计原因，采用这样的规则反而会有些痛苦。当然，实际上它也没有被广泛采用。尽管如此，我还是强烈推荐使用它。

14.2 SQL 与关系模型的不同点

此部分列出了SQL与关系模型的不同点，主要是为了参考，同时顺便进行一些附加说明。我知道可能会有人对列表中的个别术语吹毛求疵，一一解释列表中这些特性是非常不容易的，特别是它的正交性（例如，保证这些特性都相互独立，互不影响）。但是我认为这些吹毛求疵都不是重要的，重要的是它们累积起来造成的影响，坦率地说是相当惊人的[2]。

不再啰嗦了，下面具体来看一下它们的不同点：

- SQL 不能够完全区分表的值和表变量。
- SQL 表与关系（或关系变量）不同，因为它们不允许或不需要（根据具体情况而定）：（a）重复的行；（b）空值；（c）从左到右有序排列的列；（d）无名的列；（e）重复的列名；（f）指针（虽然标准中没有，但至少在某种产品中存在）；（g）隐藏的列。注意：（a）、（b）、（c）、（f）都代表破坏了信息准则。（d）和（e）（即无名的列和重复的列名）可能发生在评价某个

---

1 这个列表数据摘自 Jon Bentley 的专著 *More Programming Pearls: Confessions of a Coder* 第 9 章——*little Languages*（1988 年，Addison-Wesley 出版）。

2 这里我还要用 Wittgenstein 的名言提醒你一下：所有的逻辑差异都是巨大的差异。这一点在我和 Hugh Darwen 的一些技术专著中曾经提到过，在本书中也曾提到过。

表达式后得到的结果表中，不能出现在基表或者视图中。因此严格说来，如果我们遵守信息准则的规定（参见第 7 章），那么（d）和（e）的情况就不会破坏它，因为这样的表就不会出现在数据库中。但奇怪的是，SQL 允许我们可以得到不直接存回到数据库基表中的结果。因此，我认为（d）和（e）肯定会破坏信息准则（你可能不会相信）。

- SQL 没有恰当的表标识符。最接近于这种结构的表示方式是 VALUES 表达式（顺便说一下，处于某种特定的目的，批准或者篡改是经常使用的术语），这样的表达式就不能用在必须使用标识符的环境中。（对此，可以参见本章练习 14.2。）
- SQL 通常认为视图不是表。下面这段文字摘自 *SQL and Relational Theory*，但稍微进行了修改：

SQL 标准，以及其他大多数的 SQL 文档通常都讨论了术语“表和视图”（事实上，是相当普遍的）。显然，采用此种方式讨论的人受到了“表和视图不同”的影响，认为“表”就是基本表，基本表是物理存储的，而视图不是物理存储的。但总的来说，视图就是表（或者我更愿意把它称为关系变量）。也就是说，对规则的关系变量（至少是关系模型）执行的运算，在视图上都可以执行，因为视图就是“规则的关系变量。”

从数学的角度来说，子集就是集合。实际上，基于集合理论的操作结果得到的就是集合。这个观点同样适用于关系模型，任何关系型表达式的操作结果就是关系，特别是对于定义视图的表达式也是如此。因此，认为 SQL 中的视图不同于表的那些人是没有从关系的角度考虑问题，这有可能会引发一些错误。

- SQL 表（从前面的叙述来看，要包括视图！）至少包含 1 列。对于该点的解释，请参照附录 B（这里也说明了 SQL 不完全是关系型的）。
- SQL 没有显式的表赋值运算符。
- SQL 当然也没有显式的多变量表赋值运算（也没有类似的 INSERT/DELETE/UPDATE 操作）。
- SQL 在很多方面都破坏了赋值规则（*The Assignment Principle*），有一些是处理空值的（有一些是处理字符串类型的，还有一些是处理“可能的没有确定值的”表达式）。
- SQL 在很多方面都破坏了黄金规则（**The Golden Rule**），其中包括处理空值的，特别是推迟进行完整性检测方面。
- SQL 没有合适的“表类型”定义。
- SQL 没有表之间的“=”运算，实际上，它根本不执行表之间的比较运算。
- SQL 允许把合适的超码显式声明为码。
- SQL 的并、交、联接运算不满足交换律。

- SQL 的并、交、联接运算不满足等幂性。
- SQL 的并运算不是联接运算的特例（重复值和空值的存在是罪魁祸首）。
- SQL 中没有明确定义联接运算（原因是 SQL 表的内容是一组行的包，而不是集合）。
- SQL 没有适当的聚集运算符。
- SQL 在处理空集时会产生各种逻辑错误，包括语法和语义方面的逻辑错误。
- 很多 SQL 表的表达式都是“可能具有未确定值的”。
- SQL 支持各种行级的运算符（如游标的修改、行级的触发器等）。
- 虽然在 SQL 的标准中没有提供，但是某种特定的商业产品中使用的 SQL 专业术语有时确实会涉及物理级的构成（例如，索引）。
- SQL的视图定义中包括映射信息以及结构化信息 [1]。注意：针对该点及下一点重要性的详细讨论可参见我的一本专著：*View Updating and Relational Theory: Solving the View Update Problem*（2013 年，O'Reilly出版）。
- SQL 对于视图修改的支持是很薄弱的、针对特定问题的、不完整的。
- SQL 不能正确区分类型与描述之间的关系。（这个问题主要发生在 SQL 的子类型机制中，但不排除其他情况，详细讨论已经超出本书的范围。）
- SQL 不支持类型约束。
- SQL 的“结构化类型”有时被封装起来，有时却未被封装。注意：结构化类型是 SQL 用来自定义类型的一种机制。详细讨论也超出本书的范围。
- SQL 不能正确区分类型和类型发生器。
- 虽然SQL 标准中支持BOOLEAN 类型，但大多数商业化产品却不支持该类型。
- SQL 对于“=”的支持是存在严重缺陷的。具体来讲，“=”可以（a）比较的两个数明显不同，但也可以得到 TRUE；（b）两个比较的数没有明显差异，却不能得到 TRUE；（c）因此可以用户自定义，所以在语义上存在二义性（特别是用户自定义类型中）；（d）根本不支持定义 XML 类型；（e）虽然在标准中没有说明，但在一些特定的产品中也不支持一些特定的类型（如 BLOB 和 CLOB）。
- SQL 是基于 3-值逻辑的（可以这样说），但关系模型是基于 2-值逻辑的。
- 如前所述，SQL 不完全是关系型的。
- SQL 不直接支持映像关系，也不支持 RENAME、XUNION（排他型并运算）、MATCHING、NOT MATCHING、D_UNION、I_MINUS、D_INSERT、I_DELETE。还有其他的未在本书讨论的关系运算符。

---

1 公平地讲，这个结论也可以应用于 Tutorial D，至少在目前的情况下是可以的。

上述列出的条目并不详尽。

14.3 练习

14.1 从语义上判断下面哪些是合法的独立 SQL 表达式（即没有嵌套在其他表达式中的表达式），哪些不是？（*A* 和 *B* 是表名，假设这里的表都能够满足特定运算的需求。）注意：这个练习有点不公平，因为在本书中没有覆盖到足够的 SQL 内容来回答所有的问题，但是我想值得尝试一下，对你也会有益处的。但我至少解释一下 SQL 结构“TABLE *T*”（*T* 就是一个简单的表，而不是常用的表的表达式），它是表达式“(SELECT * FROM *T*)”的缩写。也许还要提醒你一下，在关系型中，交运算是自然联接的一种特殊形式。

a．`A NATURAL JOIN B`

b．`A INTERSECT B`

c．`SELECT * FROM A NATURAL JOIN B`

d．`SELECT * FROM A INTERSECT B`

e．`SELECT * FROM ( A NATURAL JOIN B )`

f．`SELECT * FROM ( A INTERSECT B )`

g．`SELECT * FROM ( SELECT * FROM A INTERSECT SELECT * FROM B )`

h．`SELECT * FROM ( A NATURAL JOIN B ) AS C`

i．`SELECT * FROM ( A INTERSECT B ) AS C`

j．`TABLE A NATURAL JOIN TABLE B`

k．`TABLE A INTERSECT TABLE B`

l．`SELECT * FROM A INTERSECT SELECT * FROM B`

m．`( SELECT * FROM A ) INTERSECT ( SELECT * FROM B )`

n．`(SELECT * FROM A ) AS AA INTERSECT ( SELECT * FROM B ) AS BB`

从这个练习中你可以得出什么结论？

14.2 如果 $x+y$ 是一个数值表达式，x 的值为 3，我们可以用标识符 3 代替变量引用 x，即 $3+y$（这个表达式逻辑上等价于初始的表达式）。在练习 14.1 中，如果我们用一个“表标识符”（即恰当的带有确定值的表达式）替代 *A* 或者 *B*，那么 SQL 表达式或者替代后的表达式会是什么样子的？

14.4 答案

14.1 SQL 表的表达式采用正规的 BNF 语法，为了完整地回答这个问题，可

以参照 *SQL and Relational Theory*（这个练习中的例子就摘自此书）。

a．`A NATURAL JOIN B`：不合法

b．`A INTERSECT B`：不合法

c．`SELECT * FROM A NATURAL JOIN B`：合法

d．`SELECT * FROM A INTERSECT B`：不合法

e．`SELECT * FROM ( A NATURAL JOIN B )`：合法

f．`SELECT * FROM ( A INTERSECT B )`：不合法

g．`SELECT * FROM ( SELECT * FROM A INTERSECT SELECT * FROM B )`：不合法

h．`SELECT * FROM ( A NATURAL JOIN B ) AS C`：不合法

i．`SELECT * FROM ( A INTERSECT B ) AS C`：不合法

j．`TABLE A NATURAL JOIN TABLE B`：不合法

k．`TABLE A INTERSECT TABLE B`：合法

l．`SELECT * FROM A INTERSECT SELECT * FROM B`：合法

m．`( SELECT * FROM A ) INTERSECT ( SELECT * FROM B )`：合法

n．`( SELECT * FROM A ) AS AA INTERSECT ( SELECT * FROM B ) AS BB`：不合法

至于从练习中得出的结论，要依靠你自己的回答来总结，但是我知道我自己得到的结论。

14.2　影响如下：表达式 b 原来是不合法的，但现在变成了合法的。表达式 c.、e.、k.、l.、m.是合法的，但变成了不合法的。其他所有的表达式原来是不合法的，现在仍然是不合法的。从这个练习中你可以得出什么结论？

附录 A

Tutorial D 语法

我从没用过这样大的 D。

——W. S. Gilbert：*HMS Pinafore*（1878）

该附录的目的是为 **Tutorial D** 的关系型表达式及赋值的 BNF 语法提供参考，但省略了非关系表达式的一些细节部分，特别是包含聚集运算符调用的部分，所以这里给出的是用来定义基本关系变量和约束操作的定义语法。下面的这几个部分也省略了：

- SUMMARIZE，因为该运算符有些地方被否决了（就像第 5 章解释的那样）。
- 在本书中没有讨论该语言的具体细节方面。

同时，该语法在某些方面也被简化了。特别是它没有定义映像关系，也没有使用映像关系，对操作符的优先级规则也没有过多关注。（这里提到的优先级规则，指的是语法中使用的结构，如：*r1* MINUS *r2* MINUS *r3*，计算顺序是没有明确定义的。因此需要另外定义一些语法规则来解决这些问题，但是本书中省略了这些规则。当然，可以使用括号来保证计算顺序。）下面再说明几点：

- 形如<... *name*>的所有语义解释都表示为<*identifier*>s，但具体定义没有进一步讨论。
- 虽然<*bool exp*>被忘记了，未定义，但它有助于回想起关系型比较是一种特例。
- 通常情况下，上面提到的各种逗号列表都可以为空。

A.1 表达式

```
<relation exp>
    ::= <with exp> | <nonwith exp>
```

```
<with exp>
    ::= WITH ( <temp assign commalist> ) : <relation exp>
```

注意：<temp assign>在语义上等价于<relation assign>，除非在<relvar name>出现的地方引入一个“临时性”的名字。

```
<nonwith exp>
    ::= <image relation ref> | <relation op> | ( <relation op> )

<image relation ref>
    ::= !!<relvar name> | !!( <relation exp> ) | ( <image relation ref> )

<relation op>
    ::= <relation selector> | <monadic op> | <dyadic op>

<relation selector>
    ::= RELATION [ <heading> ] { <tuple exp commalist> }
       | TABLE_DUM | TABLE_DEE
```

注意：在第一个*<relation selector>*格式中，只有在*<tuple exp commalist>*非空时，才省略可选的*<heading>*部分，TABLE_DUM 和 TABLE_DEE 分别是*<relation selector>*s RELATION { } { }和 RELATION { } { TUPLE { }}的缩写形式(参见附录 B)。

```
<heading>
    ::= { <attribute commalist> }

<attribute>
    ::= <attribute name> <type name>

<monadic op>
    ::= <relvar name> | <rename> | <where> | <project> | <extend>

<rename>
    ::= <relation exp> RENAME { <renaming commalist> }

<renaming>
    ::= <attribute name> AS <attribute name>

<where>
    ::= <relation exp> [ WHERE <bool exp> ]

<project>
    ::= <relation exp> { [ ALL BUT ] <attribute name commalist> }

<extend>
    ::= EXTEND <relation exp> : { <attribute assign commalist> }

<attribute assign>
    ::= <attribute name> := <exp>
```

注意：<*attribute assign*>有一种替换格式，在语义上等价于<*relation assign*>，除非相应的<*attribute name*>（a）出现在允许使用<*relvar name*>的地方；（b）如果讨论的属性是关系型的值，必须出现在支持目标<*relvar name*>的地方。

```
<dyadic op>
    ::= <relation exp> <dyadic op name> <relation exp>

<dyadic op name>
    ::= UNION | D_UNION | INTERSECT | MINUS | I_MINUS
       | JOIN | TIMES | MATCHING | NOT MATCHING

<relation comp>
    ::= <relation exp> <relation comp op> <relation exp>

<relation comp op>
    ::= = | ≠ | ⊆ | ⊂ | ⊇ | ⊃
```

A.2 赋值

```
<relation assignment>
    ::= [ WITH ( <temp assign commalist> ) : ]
                              <relation assign commalist> ;

<relation assign>
    ::= <relvar name> := <relation exp>
       | <insert> | <d_insert>| <delete> | <i_delete>| <update>

<insert>
    ::= INSERT <relvar name> <relation exp>

<d_insert>
    ::= D_INSERT <relvar name> <relation exp>

<delete>
    ::= DELETE <relvar name> <relation exp>
       | DELETE <relvar name> [ WHERE <bool exp> ]

<i_delete>
    ::= I_DELETE <relvar name> <relation exp>

<update>
    ::= UPDATE <relvar name> [ WHERE <bool exp> ] :
                               { <attribute assign commalist> }
```

附录 B

TABLE_DUM 和 TABLE_DEE

Tweedledum 和 *Tweedledee*
同意一战；
因为 *Tweedledum* 说 *Tweedledee*
弄坏了他漂亮的拨浪鼓。
就在这时飞来一只可怕的乌鸦
像焦油桶一样大；
这吓坏了两个英雄，
他们忘记了他们的争吵。
——摘自一个古老的英国童谣 *Through the Looking-Glass and What Alice Found There*，作者 Lewis Carroll（1871 年出版）

在第 3 章曾提到过，空集（不包含任何元素的集合）是任何集合的子集。因此空标题是一个有效的标题，空元组也是一个有效的元组，形如：TUPLE { }。实际上，有时也显式地采用 *0-tuple* 形式进行引用，这是为了强调其中没有任何组成元素，度为 0，有时也称为空元组。在 **Tutorial D** 中，其说明格式如下：

```
TUPLE { }
```

注意：在 **Tutorial D** 中，语法结构 TUPLE { }负责 2 项职责。因此，具体的语义要依据上下文环境来确定。

从上面的论述还可以进一步获知，关系也可以有空标题，如：RELATION { }，度为 0。在 **Tutorial D** 中，其说明格式如下：

```
RELATION { } body
```

其中，大括号表示空标题，*body* 可以为{ }或者{TUPLE{ }}，参见下面的解释，

也可以参见附录 A。

r 表示度为 0 的关系，那么有多少种这样的关系呢？答案是：只有 2 个。第一，当然 *r* 可能是空的（即其中不包含任何元组，回顾第 3 章练习 3.3 的答案，恰好是任一给定类型的空关系）。第二，如果 *r* 非空，那么它包含的元组必须全都为 0-tuples。但是只有一个 0-tuple（即所有的 0-tuple 都是相互重复的），所以 *r* 不可能包含多个元组。因此，实际上只有 2 个不包含属性的关系，一个只带有一个元组，另一个根本不带有元组。而且（也许令你感到惊讶），这两个关系是相当实用的，并且从理论上来说也是很重要的，以至于我们为它们另起了名字，即 TABLE_DUM 和 TABLE_DEE，或者简写为 DUM 和 DEE（DUM 代表不具有任何元组的空关系，DEE 代表只有一个元组的空关系）。

注意：Tutorial D 显式支持关键字 TABLE_DUM 和 TABLE_DEE，虽然有时它们会采用简写形式。特别是，在调用关系选择器时，TABLE_DUM 简写为：

```
RELATION { } { }
```

同样，TABLE_DEE 简写为：

```
RELATION { } { TUPLE { } }
```

（也可以参见附录 A）

TABLE_DEE 和 TABLE_DUM 这么特殊的原因是它们可以分别被解释为 TRUE 和 FALSE。顺便说明一下，考虑供应商关系变量 S 在 SNO 上的投影：

```
S { SNO }
```

令 *r* 表示该投影的结果（仍然以我们通常采用的样本数据为例，*r* 包含了 5 个元组）。现在考虑关系 *r* 在空属性集合上的投影：

```
r { }
```

显然，在根本没有任何属性的元组上进行投影将会获得空元组，因而在根本不包含属性的关系 *r* 上进行投影也会获得包含空元组的结果（空元组来自于 *r*）。但是，就像从上面例子中看到的，所有的空元组都是相互重复的。因此，在不包含任何属性的关系 *r*（就像上面定义的一样）上进行投影将获得不包含任何属性的关系，而且只有一个（空）元组，即 TABLE_DEE。

题外话：前面的段落说明了这样一个事实（我认为是显而易见的），定义一个可以用于元组而不是关系的类似的投影运算是有意义的。具体内容如下：元组 t 具有标题 H，X 是 H 的子集，那么 t 在 X 上的（元组）投影是一个具有标题 X 的元组，即 t 的子集是由与 X 中提

到的属性相一致的 *t* 中的构件组成的。**注意**：使得类似的元组定义有意义的其他关系运算符包括 EXTEND、RENAME、UNION。进一步的讨论可以参见 *SQL and Relational Theory*。

回顾第 6 章（特别是练习 6.4 的答案），每个关系变量（实际上是每个关系表达式）都有相应的断言。对于关系变量 S，其断言形式可以如下：

供应商 SNO 受到契约的约束，其名字为 SNAME，状态为 STATUS，所在城市为 CITY。

S 在 SNO 上的投影 *r*，可以表示如下：

存在某个姓名 SNAME、某个状态 STATUS、某个城市 CITY，其供应商 SNO 是受到契约约束的，其名字为 SNAME、状态为 STATUS、所在城市为 CITY。

属性为空的 *r* 的投影 *r*{ }，可以表示如下：

存在某个供应商号码 SNO、某个供应商名字 SNAME、某个供应状态 STATUS、某个供应城市 CITY，其对应的供应商 SNO 是受到契约的约束的，其名字为 SNAME、状态值为 STATUS、所在城市为 CITY。

观察最后一个断言，它实际上是一个命题，虽然它乍看上去有些复杂。它无条件地评定为TRUE或者FALSE[1]。在目前情况下，*r*{ }是TABLE_DEE，显然该断言（或命题）评定为TRUE。但假设此时数据库中根本没有供应商存在，那么关系变量S将是空的，投影S{SNO}将得到一个空关系*r*，投影*r*{ }将是TABLE_DUM，问题中的断言（命题）将评定为FALSE。

所以，TABLE_DUM 就是 FALSE（或者 *no*），TABLE_DEE 就是 TRUE（或者 *yes*）。这是它们最基本的含义！顺便提一下，记住它们最好的方法就是：DEE 和 *yes* 都有一个“E”，但 DUM 和 *no* 没有。

现在，我们已经陷入了某些事情的一些细节的困境，这已经远远超出了本书的范围，所以现在我要给出一些解释。首先，给出一个 DEE 和 DUM 操作的具体例子，考虑查询“位于雅典的每个供应商都供应零件了吗”？这是一个 *yes/no* 的查询（也就是说，答案或者是 *yes*，或者是 *no*）。**Tutorial D** 的语法格式如下：

```
( ( S WHERE CITY = 'Athens' ) MATCHING SP ) { }
```

因为这个表达式最终是在无属性的关系上进行投影，显然会得到 TABLE_DUM 或者 TABLE_DEE。如果结果为 TABLE_DUM，则表示 no，即在雅典没有供应商供应了任何零件；如果结果为 TABLE_DEE，则表示 yes，即雅典的供应商至少供

---

1 实际上，如果断言中的参数集合恰好为空集，则断言通常就变为了命题。注意，这种情况仅限于目前讨论的例子中。具体来讲，在投影 *r*{ }的断言中，符号 SNO、SNAME、STATUS、CITY 不再表示参数本身，相反它们表示的是约束变量（参见附录 D）。

应了一种零件。

那么用SQL如何表示呢？SQL中规定得非常明确，任何查询的结果都是表。但不幸的是，它忘记了支持的表要表达的含义是yes或者no[1]。所以，在SQL中不能直接完成yes/no的查询。实际上，针对上面的查询实例，在SQL中最好采用如下的方式实现。首先，询问雅典的供应商提供了多少零件：

```
SELECT COUNT ( SNO ) AS X
FROM   S
WHERE  CITY = 'Athens'
AND    SNO IN
     ( SELECT SNO
       FROM   SP )
```

然后，从该查询得到的结果表中针对每一行抽取出列 X 的取值。如果该值为 0，则表示雅典没有供应商来供应任何零件，如果该值为非 0，则表示雅典至少有 1 个供应商至少供应了 1 种零件。

我要强调的另外一点是（在某种程度上，它是更通用的、更基本的）：两个特殊的关系 TABLE_DUM 和 TABLE_DEE（特别是 TABLE_DEE）在关系代数中所起到的作用类似于传统算术中 0 的作用。我们都知道 0 的重要作用。实际上，很难想象，算术中没有 0 的话会是什么样（古罗马人曾经试图做过类似的事情，但是也没有坚持多久）。因此，在关系代数中如果没有 TABLE_DEE 的话，也是难以想象的。

对于前面叙述的内容我再给出一些特殊说明。首先，众所周知，在逻辑运算中TRUE相对于AND运算是恒等的（我在第 5 章曾提到过），也就是说，如果*p*是任意一个命题，那么表达式*p* AND TRUE 和TRUE AND *p*都被简化为*p*。同样，在普通的算术中，0 与“+”等价，1 与“*”等价，即对于所有的数字*x*，表达式$x + 0$、$0 + x$、$x * 1$、$1 * x$都可以简化为*x*。类似地，在关系代数中，TABLE_DEE与JOIN等价，即TABLE_DEE与任何关系*r*的联接运算都简化为*r*[2]。因此，特殊情况下，因为没有命题的AND运算结果为TRUE，没有数字的求和运算结果为 0，没有数字的乘积运算结果为 1，所以没有关系的联接运算结果为TABLE_DEE。

我再强调一下，你现在已经知道了存在这两个特殊关系，而且你会发现它们可

---

1 它也不支持空行（绝大部分情况下不支持，但不是一直不支持），详细解释请参见 *SQL and Relational Theory*。

2 就像我在第 4 章解释的，如果操作数没有公共的属性，那么联接运算就退化为笛卡儿乘积。如果操作数之一为 TABLE_DEE 或者 TABLE_DUM，则联接运算也退化为笛卡儿乘积。我们正在讨论的联接运算实际上就是笛卡儿乘积。

以应用在任何地方。实际上，它们只是 nullology 概念的特殊而重要的一种表现形式，这些内容我在第 3 章的脚注中都有简要说明。注意：我在脚注中也讲过，nullology 与 SQL 的类型 null 无关！Nullology 研究的是空集，理论上严格，而且相当实用。SQL 的 null 理论上不是很严格，而且使用时也要避免，以免造成麻烦。

最后，或许我应该再讨论一点 TABLE_DUM 和 TABLE_DEE。首先，对于非英语国家的读者来说，它们只是 *Tweedledum* 和 *Tweedledee* 的一种文字游戏（就像该附录中开头的题辞一样），这是英国一首童谣中的两个人物，摘自作家 Lewis Carroll 的 *Through the Looking-Glass*。其次，这两个名字或许有一点可悲，仅因为这两个关系不能被合理地描述为关系！（注意，在前面的讨论中，我甚至没有试图去勾勒这两个关系的画面。实际上，只是考虑的关系的概念，而表的概念已经被完全破坏了。）但是在关系型世界里，使用这两个名字已经很长时间了，也许我们也不能去改变它们。

附录 C

集合论

集合的乐趣。

——Anon：*Where Bugs Go*

理解集合理论可有助于处理数据库。实际上，就像第 7 章提到的，关系模型是直接建立在这个理论的某些特定方面基础上的。因此，在本附录中，我要简要介绍一下集合理论的基础知识。

C.1 什么是集合

首先给出一个定义：

定义：**集合** *S* 是有一组不同的**元素**构成的，换句话说，它的成员就是给定任一对象 *x*，判断 *x* 是否包含在 *S* 中（即是 *S* 中的一个元素）。注意这个术语：集合可以说成是**包含**其成员。

下面给出一个例子：

```
{ 2 , 3 , 5 , 7 }
```

这个例子是标准的书面表示格式，元素封装在大括号内，利用逗号分隔。强调下面几点。

- 集合中不能包含重复元素。因此，如果书面的表示中出现重复的元素（按照惯例，通常不会出现这种情况），重复的元素往往会被忽略，例如{ 2 , 2 , 3 , 5 , 5 , 5 , 7 , 7 }（尽管不可能），另一种书面的表示为{2,3,5,7}。
- 集合中的元素没有顺序，因此，例如{ 7 , 2 , 5 , 3 }，在书面上可能会有另一种表示形式。
- 不包含任何元素的集合称为空集，它是唯一的，记作{}，有时也记作 Ø。

- 还有另一个唯一的集合，称为全集，即包含了所有感兴趣的元素（所有感兴趣的元素集合，可以是讨论问题的上下文环境中涉及的所有数据。如前面例子中的所有正整数。）有时也称为问题域。
- 集合中的元素可以代表任何事物，甚至可以是其他集合。注意：这种描述有点过于简单了，但是也足以达到该附录的目的。
- 集合中元素的数量称为集合的度。注意，空集的度为 0。

目前，存在多种集合的表示方法。第一，枚举法，即一一列出集合中的元素。如：

```
{ 2 , 3 , 5 , 7 }
```

然而，枚举法只适用于有限集合，但无限集合也是允许存在的。然而，在计算环境下，集合必须是有限的。所以在本附录中对于无限集合不做过多的讨论。

第二，断言法（或谓词法）。例如：

```
{ x : x is a prime AND x < 10 }
```

该规则定义了由所有对象 x 组成的集合，x 必须为质数而且小于 10（规则中的冒号“:”可以读作：“就是”）。当然，该例中的集合与上面采用枚举法表示的集合是相等的。

第三，替代法。例如：

```
{ x² : x is a prime AND x < 10 }
```

该规则定义了由所有对象 y 组成的集合，$y=x^2$，x 就是小于 10 的质数。换句话说，该集合中的元素为{4，9，25，49}。注意：采用断言法表示的规则当然可以看作是采用替代法表示规则的一种特例。

现在来定义集合的成员关系运算符“∈”（该符号来自于希腊字符的第 5 个字母，有时也用 *epsilon* 进行引用）。

定义：表达式 $x \in S$ 值为 TRUE，当且仅当 x 是集合 S 的成员。相反，表达式 $x \notin S$ 返回值为 FALSE，当且仅当 x 不是集合 S 的成员。注意：表达式读作“x 出现在 S 中”，或者“x 属于 S”，或者“x 是 S 的成员”。当然，表达式 $x \notin S$ 的读法也是类似的。

例如，$5 \in \{2,3,5,7\}$返回值为 TRUE，而 $8 \in \{2,3,5,7\}$的返回值为 FALSE。注意：在对象 x 和单元素集合（singleton set）$\{x\}$（表示只包含 1 个元素的集合）是有区别的。

下面以另外一个定义来结束本部分的介绍。

定义：集合论是数学的一个分支，与断言（谓词）逻辑有密切关系，主要目的是处理集合的特性。特别是它依据已有的公理规范定义了集合的概念，例如，**外延**

公理。该公理说明：当且仅当两个集合恰好有完全相同的元素时，两个集合才相等。

该理论的等价定义起到一个很关键的作用。首先，集合成员关系的定义要依赖于它（别忘了，如果没有测试 5 是否与集合{2,3,5,7}中的某个元素相等的能力，我们如何判断表达式 5∈{2,3,5,7}是否返回 TRUE。其次，反过来外延公理也依赖于它（这是显然成立的）。

C.2 子集和超集

假设 *S1* 和 *S2* 都是集合，二者不是必然不同的。那么就可以进行如下定义：

定义：*S2* 是 *S1* 的子集（符号表示：*S2*⊆*S1*），当且仅当 *S2* 中的每一个元素都存在于 *S1* 中。*S1* 是 *S2* 的超集（符号表示：*S1*⊇*S2*），当且仅当 *S2* 是 *S1* 的子集。

强调下面几点。

- 集合的每个子集都是一个集合，每个超集也都是集合。
- 每个集合既是自身的子集，也是自身的超集。
- 空集是任一集合的子集，全集是每个集合的超集。

下面的定义与上面定义是等价的，但是介绍了一些新的术语。

定义：*S2* 包含于 *S1*（符号表示：*S2*⊆*S1*），当且仅当 *S2* 中的每一个元素都存在于 *S1* 中。*S1* 包含 *S2*（符号表示：*S1*⊇*S2*），当且仅当 *S2* 被包含于 *S1* 中。

强调下面几点。

- 每个集合都包含其自身，也被包含于本身中。
- 空集被包含在每个集合中，全集包含每个集合。
- 通常采用术语集合包含（*set inclusion*）来指代“⊆”，而不是“⊇”(稍微有点乱)。
- 术语：不要混淆“⊆”和“∈”！一个集合包含子集，但包括它的元素。然而要注意的是，这个标识符也不是完全一致的。实际上，至少在英语表示中有时很难保持它的一致性，因为偶尔会有写作风格上的一些惯例，要求采用“错误的”的单词。读者要擦亮眼睛。
- 现在可以定义集合 *S2* 和 *S1* 的等价，当且仅当表达式 *S1*⊆*S2* 及 *S1*⊇*S2* 都返回 TRUE 时，则 *S2*=*S1*。

我们说 *S2* 是 *S1* 的子集并不排除 *S2* 和 *S1* 是同一个或者是相等的集合。如果我们想排除这种可能性，就必须要讨论真子集。下面给出相应的定义。

定义：*S2* 是 *S1* 的真子集（符号表示：*S2*⊂*S1*）（等价的说法：*S2* 真包含于 *S1*），当且仅当 *S2* 是 *S1* 的子集，*S2* 和 *S1* 是不同的（即 *S1* 的某个元素不在 *S2* 中）。*S1* 是 *S2* 的真子集（符号表示：*S1*⊂*S2*）（等价的说法：*S1* 真包含于 *S2*），当且仅当 *S2*

是 *S1* 的真子集。

例如，*S1* 为{2,3,5}，*S2* 为{2,5}，那么 *S2* 是 *S1* 的真子集，它也是自身的子集，但不是真子集（集合都不是自身的真子集）。

定义：给定集合 *S* 的幂集 *P*(*S*)是集合 *S* 的所有子集构成的集合。

例如：假设集合 *S* 为{ 2 , 3 , 5 }，则其幂集 *P*(*S*)为{ { } , { 2 } , { 3 } , { 5 } ,{ 2 , 3 } , { 3 , 5 } , { 5 , 2 } , { 2 , 3 , 5 } }。

这个幂集的度为 8（$=2^3$）。通常情况下，假设集合 *S* 的度为 *n*，则幂集 *P*(*S*)的度为 2^n（不要忘了，空集和集合 *S* 本身都是 *S* 的子集）。

C.3　练习

设集合 *S* 为{ 2 , 3 , { 2 , 5 } , 5 , { 3 , 7 } , 7 , { 2 , { 3 , 7 } } }

C.1　*S* 的度为多少？

C.2　表达式{3,7}∈*S* 的值？

C.3　表达式{3,7}⊆*S* 的值？

C.4　表达式{2,3,7}∈*S* 的值？

C.5　表达式{2,3,7}⊆*S* 的值？

C.6　表达式{ }⊆*S* 的值？

C.7　幂集 *P*(*S*)的度为多少？

C.8　存在使得 *P*(*X*) = *X* 成立的集合 *X* 吗？

C.4　答案

C.1　7

C.2　TRUE

C.3　TRUE

C.4　FALSE

C.5　TRUE

C.6　TRUE（这是必然成立的）

C.7　2^7=128

C.8　不存在。假设 *X* 的度为 *n*，则 *P(X)*的度为 2^n。因为 2^n 总是大于 *n*（即使 *n*=0），*P(X)*与 *X* 不会相等。因此，该题的答案为不存在，即使 *X* 是空集。注意：如果 *n*=0，$2^n=2^0=1$。

C.5 集合运算符

大多数读者都熟悉集合论中的并运算和交运算[1]，但在这里从记录的角度来定义它们。

定义：集合 *S2* 和 *S1* 的**并**（符号表示：*S1* ∪ *S2*，有时读作“*S1*cup *S2*”)是由对象 *x* 构成的集合，*x* 或者是 *S1* 的元素，或者是 *S2* 的元素，或者同时为 *S1* 和 *S2* 的元素，也可以表示为：$\{x : x \in S1 \text{ OR } x \in S2\}$。

定义：集合 *S2* 和 *S1* 的**交**（符号表示：*S1* ∩ *S2*，有时读作“*S1*cap *S2*”）是由对象 *x* 构成的集合，*x* 同时为 *S1* 和 *S2* 的元素，也可以表示为：$\{x : x \in S1 \text{ AND } x \in S2\}$。

集合的差运算也与此相似。

定义：集合 *S2* 和 *S1* 的**差**（按照定义顺序表示为：*S1* – *S2*），是由对象 *x* 构成的集合，*x* 为 *S1* 的元素，但不是 *S2* 的元素。即：$\{x : x \in S1 \text{ AND } x \notin S2\}$。

从前面的定义可以看到，集合论的并运算和交运算分别对应于逻辑运算符 OR 和 AND。差运算对应于逻辑运算符 NOT。与补运算相应的定义如下。

定义：集合 *S* 的**补**（符号表示：~*S*，有时读作“not *S*”）是由对象 *x* 构成的集合，*x* 不存在于集合 *S* 中。表示为：$\{x : x \notin S\}$。

然而，如果 *U* 是全集，那么 *S* 的补集就是差运算 *U*−*S*。实际上，差运算 *S1*−*S2* 有时也成为 *S2* 与 *S1* 的相对补。*U*−*S* 称为集合 S 的绝对补。

另一个有用的运算符是异或运算，也称为对称差，与逻辑运算符 XOR 对应（排他性的或）。

定义：集合 *S2* 和 *S1* 的异或是由对象 *x* 构成的集合，*x* 或者是 *S1* 的元素，或者是 *S2* 的元素，但不同时是 *S1* 和 *S2* 的元素。表示为：$\{x : x \in S1 \text{ XOR } x \in S2\}$。

注意：目前在异或运算符的符号表示上还未达成一致意见。部分原因是在附录的其余部分，我要使用关键字来表示各种运算符，如：UNION、INTERSECT、MINUS（差运算或相对补）、COMP（绝对补）和 XUNION（异或运算或对称差）。

C.6 练习

假设 *S1* 为{2,3,5,7}，*S2* 为{1,3,5,7,9}，请计算下列表达式的值。

C.9 `S1 UNION S2`

C.10 `S1 INTERSECT S2`

C.11 `S1 MINUS S2`

---

1 参见第 4 章练习 4.5。

C.12 *S2* MINUS *S1*

C.13 *S1* XUNION *S2*

C.14 *S1* MINUS (*S1* MINUS S2)

C.15 (*S1* MINUS *S2*) UNION (*S2* MINUS *S1*)

C.16 COMP (*S1*)

C.7 答案

C.9 {1,2,3,5,7,9}

C.10 {3,5,7}

C.11 {2}

C.12 {1,9}

C.13 {1,2,9}

C.14 {3,5,7}（与 C.10 类似）

C.15 {1,2,9}（与 C.13 类似）

C.16 该题的答案要依赖于全集，为了简化问题，假设全集为 1 至 9 的整数构成的集合，则 COMP(*S1*)的结果为{1,4,6,8,9}。

C.8 一些特性

至此，在该附录中我们已经讨论了除“∈”之外的所有运算符，这些运算符的输入和输出都是集合，即我们是在一个封闭的系统中来讨论这些运算符的。除此以外，还有一些特性，具体如下 [1]。

- **交换律**：二元运算符*Op*具有交换性，当且仅当对于所有的*A*和*B*，满足 $A\ Op\ B \equiv B\ Op\ A$[2]。集合运算符UNION、INTERSECT、XUNION都满足交换律。
- **结合律**：二元运算符 *Op* 具有交换性，当且仅当对于所有的 *A*、*B* 和 *C*，满足 $A\ Op\ (B\ Op\ C) \equiv (A\ Op\ B)\ Op\ C$。集合运算符 UNION、INTERSECT、XUNION 都满足结合律。

---

1 大部分的特性都可以采用文氏图的形式来表示（留给读者练习）。注意：如果你不熟悉文氏图的定义，可以在关于逻辑学的基础知识书中找到解释，例如可以参见 Robert R. Stoll 的 *Set Theory and Logic*（1965 年由 W. H. Freeman and Company 出版，1979 年由 Dover Publications 再版）。

2 这里采用中缀语法风格，该部分始终采用这种风格。同时，符号“≡”读作“恒等于”。

- **分配律**：二元运算符 *Opx* 可以分配给二元运算符 *Opy*，当且仅当对于所有的 *A*、*B* 和 *C*，满足 *A Opx* (*B Opy C*) ≡ (*A Opx B*) *Opy* (*A Opx C*)，集合运算符 UNION 和 INTERSECT 都相互满足分配律。
- **幂等律**：二元运算符 *Op* 满足幂等律，当且仅当对于所有的 *A*，满足 *A Op A*≡*A*。集合运算符 UNION 和 INTERSECT 满足幂等律。
- **吸收律**：二元运算符 *Opx* 吸收二元运算符 *Opy*，当且仅当对于所有的 *A* 和 *B*，满足 *A Opx* (*A Opy B*) ≡ *A*。集合运算符 UNION 和 INTERSECT 项目满足吸收律。

同时，还具有如下性质：

- 德摩根定律

```
COMP ( A UNION B ) ≡ ( COMP ( A ) ) INTERSECT ( COMP ( B ) )
COMP ( A INTERSECT B ) ≡ ( COMP ( A ) ) UNION ( COMP ( B ) )
```

- 假设 *U* 是全集，则满足如下性质：

```
A UNION U ≡ U
A INTERSECT U ≡ A
A UNION ∅ ≡ A
A INTERSECT ∅ ≡ ∅
```

C.9　练习

证明下列恒等式成立或者不成立：

C.17　(*A* MINUS *C*) MINUS (*B* MINUS *C*) ≡ (*A* MINUS *B*) MINUS *C*

C.18　(*A* INTERSECT *B*) UNION *C* ≡ *C* INTERSECT (*B* UNION *A*)

C.19　*A* INTERSECT (*B* MINUS *C*) ≡ (*A* INTERSECT *B*) MINUS (*A* INTERSECT *C*)

C.10　答案

我给出练习C.17 的答案，只是为了说明这样的问题如何进行证明的过程 [1]。

C.17　该恒等式是有效的，证明如下：

```
x∈( ( A MINUS C ) MINUS ( B MINUS C ) ) 当且仅当
```

1　也可以采用文氏图表示。

```
x∈ ( A MINUS C ) AND x ∉ ( B MINUS C ) 当且仅当
x∈A AND x ∉ C AND ( x ∉ B OR x∈C ) 当且仅当
x∈A AND ( x ∉ C AND ( x ∉ B OR x∈C ) ) 当且仅当
x∈A AND ( ( x ∉ C AND x ∉ B ) OR ( x ∉ C AND x∈C ) ) 当且仅当
x∈A AND ( x ∉ C AND x ∉ B ) 当且仅当
x∈A AND x ∉ B AND x ∉ C当且仅当
x∈ ( A MINUS B ) AND x ∉ C当且仅当
x∈ ( A MINUS B ) MINUS C
```

C.11 集合代数

下列条件放在一起构成了集合代数：

（a）集合 U 定义为给定集合 S 的幂集 $P(S)$；

（b）运算符 UNION、INTERSECT、COMP 可以作用于 U 中的元素；

（c）包含运算符“⊆”可以作用于 U 中的元素。

在集合代数中，全集 U 起着很重要的作用。另外集合代数还满足如下规则：

- 闭包性（适用于 UNION、INTERSECT、COMNP）
- 交换律（适用于 UNION、INTERSECT）
- 结合律（适用于 UNION、INTERSECT）
- 分配律（适用于 UNION、INTERSECT，相互满足）
- 同一律（UNION 和 INTERSECT 对于 Ø 和 U 满足同一律）
- 幂等律（适用于 UNION、INTERSECT）
- 吸收律（适用于 UNION、INTERSECT，相互满足）
- 回归律 (COMP (COMP(A)) ≡ A
- 互补律 A UNION (COMP(A)) ≡ U A INTERSECT (COMP(A)) ≡ Ø
- 德摩根定律（参见前面的叙述）
- 恒等律，下面这些表达式永远成立：

```
A⊆B
A UNION B = B
A INTERSECT B = A
```

C.12 笛卡儿乘积

下面开始讨论集合理论是如何为关系理论服务的（但仅仅是开始，我故意针对这个问题不想讨论太多）。

定义：有序对<*x*,*y*>由两个元素*x*和*y*构成的，*x*称为第一元素，*y*称为第二元素。更准确地讲，<*x*,*y*>是集合{{*x*},{*x*,*y*}}的简写。该集合中的元素即决定了有序对中的元素，也决定了元素的顺序[1]。

从该定义中可以得到如下定义，有序对<*x1*,*y1*> 和 <*x2*,*y2*>相等，当且仅当 *x1* = *x2* 且 *y1* = *y2*（因此，通常情况下，<*x*,*y*> ≠ <*y*,*x*>）。

现在来定义笛卡儿乘积[2]：

定义：集合 *S1* 和 *S2* 的笛卡儿乘积（注意顺序）是由所有形如<*x*,*y*>的有序对构成的，其中 *x* 属于 *S1*，*y* 属于 *S2*。

注意，集合 *S1* 和 *S2* 的笛卡儿乘积实际上就是一个二元关系，因此，有如下定义。

定义：集合 *S1* 和 *S2* 的二元关系（注意顺序）是集合 *S1* 和 *S2* 的笛卡儿乘积的子集。

举个例子，假设*S1* 为{1,3}，*S2* 为{2,3,5}，则集合*S1* 和*S2* 的二元关系如下（只给出了四种形式）[3]：

```
{ <1,2> , <3,3> , <3,5> }
{ <1,2> , <1,3> , <1,5> , <3,2> , <3,3> , <3,5> }
{ <1,5> }
{ }
```

给出最后一个定义：

定义：函数就是二元关系，其中没有任何两个元素的第一组成部分是相同的。

C.13 总结

下面对于集合论给出一些总结。如你所见，我们已经涉及了很多内容，这些内容对于理解目前的内容是足够的，但是当然还有很多内容。我希望这些足以激起你的兴致。如果你还想了解更多，我建议你阅读曾在该附录中提到的那本书（见脚注2），即《集合论与逻辑》，作者 Robert R. Stoll（（1965 年由 W. H. Freeman and Company 出版，1979 年由 Dover Publications,再版。），当然也可以阅读与内容相关的其它材料。

---

1 你可能认为这个定义破坏了一种情况，即元素 *x* 和 *y* 相等（因为表达式{{*x*},{*x*,*y*}}就变成了{{*x*}}）。但实际上不会破坏，详细原因已经超出了本书的范围。

2 集合论的笛卡儿乘积与关系模型的笛卡儿乘积不完全一样。因此，集合论的二元关系与关系模型也不完全一样。详细讨论参见 *SQL and Relational Theory* 一书中的附录 A。

3 作为练习，给出这四种关系的表格形式。

附录 D
关系演算

如果它是这样，就可能是；
如果它已经这样，就将是；
但当它不是时，它就不是。
那就是逻辑。

——Lewis Carroll：*Through the Looking-Glass and What Alice Found There*
（1871 年出版，*Tweedledee*）

就像第 7 章提醒注意的一样，关系演算相当于关系代数的替代品，在表示方式上是等价的。它建立在断言算术的基础上（也被称作是断言逻辑），实际上，它是断言算术的一个剪裁版本，专门用来处理关系。而且，关系模型的相关知识也不能说是非常完整的，因为它至少没有包含算术的基本知识。因此，在该附录中，针对这个主题给出一个简单的说明。

该附录的计划如下：依照这些介绍性的评论，将展示一系列简单的例子，这些例子主要涉及查询和约束，目的是给出形如“操作”的算术表达式的核心思想和表示形式，同时给出这些表达式的语法。然而请注意，我没有在该附录中描述商业上使用的语言，也没有描述任何与工业应用相近的事情。实际上，我只是给出并使用一种假设层面的语法，但这足以起到说明的目的。

算术的一个最基本特征就是**域变量**。简言之，域变量就是在某些特定关系范围内的变量（即允许变量的值取自那个关系中的元组）[1]。所讨论的关系一般都是特定关系变量的当前值。因此，如果定义域变量*RV*的取值范围为关系*R*，那么在任一给定时间，表达式*RV*（实际上是域变量的引用）表示在关系*r*范围内的某些元组，

1 你可能会记得，前面我们曾遇到过域变量，尽管只是在 SQL 的环境下（参见第 11 章练习 11.1d 的答案）。特别提醒，域变量不是通常编程语言中使用的变量，相反，它是逻辑层面的变量。

关系r恰好是讨论的关系R的当前取值。例如，考虑"查询伦敦供应商的供应商号码"，采用关系演算表示的语法如下：

```
RANGEVAR SX RANGES OVER S;
{ SX.SNO } WHERE SX.CITY ='London'
```

该表达式中有一个域变量SX，它的范围是供应商变量S的值，该值由表达式第二行的关系演算表示计算得到。该表达式可以读作（不严格地讲，这种解释有点生硬）："对于域变量SX中的每一个可能的元组（换句话说，是当前出现在关系变量S中的每个元组），返回该元组的SNO值，当且仅当该元组的CITY值为伦敦。"

术语：把WHERE子句前面的部分称为原元组（*proto tuple*），就像前面关系演算表达式例子中的第二行，因为在计算整个表达式值的时候，它代表了出现在结果值中的元组的原型[1]。所以，关系演算的语法格式如下：

```
proto tuple WHERE bool exp
```

该表达式将对包含特定元组的关系进行取值，这些特定元组的值为通过布尔表达式所确定的原元组的取值，而布尔表达式的值必须由WHERE子句判断为TRUE。

通过前面的介绍，接下来给出一系列实例（查询实例和约束实例）来加深对内容的理解。注意：为了简化，在大部分表达式中省略了域变量定义（例如，RANGEVAR说明），但在实际情况下必须使用域变量。作为替代，假设域变量SX、SY等定义在关系变量S上，域变量PX、PY等定义在关系变量P上，域变量SPX、SPY等定义在关系变量SP上。而且，还对有必要说明的一些例子给出了详细的解释，这些例子的结果都不能直观地得出。

D.1 查询实例

1. 查询状态值大于20的巴黎供应商的号码和状态值。

```
{ SX.SNO , SX.STATUS } WHERE SX.CITY = 'Paris' AND SX.STATUS > 20
```

2. 查询位于同一城市的供应商x和y的供应商号码对。

```
{ XNO := SX.SNO , YNO := SY.SNO }
                     WHERE SX.CITY = SY.CITY AND SX.SNO < SY.SNO
```

1 术语**原元组**（*proto tuple*），用在这里是很恰当的，但不是标准的。实际上，目前还没有一个标准的术语来表示该部分的名称。

3．查询提供零件 P2 的供应商的所有信息。

```
{ SX.SNO , SX.SNAME , SX.STATUS , SX.CITY }
          WHERE EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = 'P2' )
```

解释：如你所见，WHERE 子句中的布尔表达式（称为 *bx1*）可以采用如下形式表示：

```
EXISTS RV ( bx2 )
```

RV 是域变量，EXISTS *RV* 是一个量词（quantifier，确切地说是存在量词），*bx2* 是另一个布尔表达式。整个布尔表达式 *bx1* 的返回值为 TRUE，当且仅当至少存在一个元组（该元组位于外部，可能值来自 *RV*）使得内部的布尔表达式 *bx2* 返回值为 TRUE。因而，我们可以想象这个例子中的查询执行过程如下：

- 系统会检测关系变量 S 的当前值中的元组（即关系变量 S，因为原元组涉及域变量 SX，它的取值范围就是关系变量 S），每次检查一个元组，顺序是任意的。
- 考虑这样的一个元组，即供应商 S1 的元组，WHERE 子句中的布尔表达式就替换为如下形式：

```
EXISTS SPX ( SPX.SNO = 'S1' AND SPX.PNO = 'P2' )
```

显然，当且仅当关系变量 SP 当前包含供应商 S1 和零件 P1 时，该表达式的返回值为 TRUE。样本数据仍然按照本书中使用的数据，因此关系变量 S 中供应商 S1 的元组就成为最终结果的一部分。

- 然后，系统就转向去判断关系变量 S 的另一个元组，重复这个过程直到所有的元组都被检测完毕。

针对这个例子，再强调以下几点：

- 符号表示问题：EXISTS 通常采用反写的 E 表示，即 ∃，但在该附录中仍然使用该关键词，即 EXISTS。
- **术语**：被量化的域变量要受到问题中量词的限制（即域变量所出现的整个演算表达式），不受限制的域变量被称作自由变量。如该例子中的 SPX 就是受限的，而 SX 就是自由的。
- 该例中的原元组实际上包含了相应关系变量的所有属性，为了简化，可以把这样的原元组简写成如下形式：

```
{ SX } WHERE EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = 'P2' )
```

4．查询至少供应了一个红色零件的供应商姓名。

```
{ SX.SNAME } WHERE EXISTS SPX ( SX.SNO = SPX.SNO AND
                   EXISTS PX ( PX.PNO = SPX.PNO AND PX.COLOR = 'Red' ) )
```

或者等价地表示为（但是在前束范式中，所有的量词必须出现在 WHERE 子句的布尔表达式之前）：

```
{ SX.SNAME } WHERE EXISTS SPX ( EXISTS PX ( SX.SNO = SPX.SNO AND
                          PX.PNO = SPX.PNO AND PX.COLOR = 'Red' ) )
```

无论哪一种方式，该例中要注意的一点是出现在 EXISTS *RV* (*bx*)量化表达式中的布尔表达式 *bx* 自身涉及再次量化的问题。

5．查询至少供应了一种由供应商 S2 提供的零件的供应商姓名。

```
{ SX.SNAME } WHERE EXISTS SPX ( EXISTS SPY ( SX.SNO = SPX.SNO AND
                                             SPX.PNO = SPY.PNO AND
                                             SPY.SNO = 'S2' ) )
```

观察一下，这个表达式中有两个不同的域变量，但取值范围是相同的。注意：实际上我们可能想在该例中 WHERE 子句的布尔表达式中加入如下成分（但确切的原因是什么呢？）：

```
AND SX.SNO ≠'S2'
```

6．查询供应了所有零件的供应商姓名。

```
{ SX.SNAME } WHERE FORALL PX ( EXISTS SPX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

解释：如你所见，WHERE 子句的布尔表达式（暂且称作 *bx1*）可以表示为如下形式：

```
FORALL RV ( bx2 )
```

同 EXISTS *RV* 一样，FORALL *RV* (*bx2*)也是量词（专门称作全称量词）。整个布尔表达式 *bx1* 返回值为 TRUE，当且仅当取自 *RV* 的元组集合中的每个元组都使得布尔表达式 *bx2* 的返回值为 TRUE，因此我们可以想象一下该例的具体执行过程：

- 系统检测关系变量 S 当前值的所有元组，一次检查一个，次序是任意的。
- 假设考虑供应商 S1 的元组，WHERE 子句中的布尔表达式就可以替换为如下形式：

```
FORALL PX ( EXISTS SPX ( SPX.SNO = 'S1' AND SPX.PNO = PX.PNO ) )
```

当且仅当关系变量 SP 当前包含了供应商 S1 以及零件号为 *Px* 的元组（每个零件号 *Px* 都会出现在关系变量 P 中的当前值中），则该表达式的返回值为 TRUE。仍然采用本书中的样本数据，因此关系变量 S 的供应商 S1 的元组就成为最终结果的一部分。

- 然后系统继续检测关系变量中的另一个元组，这个过程反复进行，直到所有这样的元组都被检测完毕。

针对该例，再强调几点：

- 符号表示问题：FORALL 通常采用倒写的 A 表示，即∀，但在该附录中仍然采用关键词 FORALL。
- 一个“好的”例子足以使存在量词表达式的返回值为 TRUE，然而一个“坏的”例子足以使一个全称量词表达式的返回值为 FALSE。
- 任何涉及FORALL的布尔表达式都可以转化为采用EXISTS的形式[1]，规则如下：

```
FORALL X ( bx ) ≡ NOT EXISTS X ( NOT bx )
```

因此第 6 个例子可以替换为如下形式：

```
{ SX.SNAME } WHERE NOT EXISTS PX ( NOT EXISTS SPX
                                    ( SPX.SNO = SX.SNO AND
                                      SPX.PNO = PX.PNO ) )
```

题外话：前面的讨论对于SQL有专门的规则，因为SQL不支持FORALL，虽然它支持EXISTS[2]。这种情况的一种结果就是看上去自然用FORALL实现对查询可能很难在SQL中表示。针对这种情况，可以参见第 11 章练习 11.1e的答案。

7．查询没有供应零件 P2 的供应商姓名。

```
{ SX.SNAME } WHERE NOT EXISTS SPX
                        ( SPX.SNO = SX.SNO AND SPX.PNO = 'P2' )
```

8．查询至少供应了由供应商 S2 提供的所有的供应商号码和所在城市。

```
{ SX.SNO , SX.CITY } WHERE FORALL SPX ( SPX.SNO ≠'S2' OR
                             EXISTS SPY ( SPY.SNO = SX.SNO AND
                                          SPY.PNO = SPX.PNO ) )
```

解释：查询供应商 SX 的号码和所在城市，而 SX 要满足对于所有的供应关系

---

1 当然，反过来也是可以的。

2 实际上可以说，它对 EXISTS 的支持也是有缺陷的（参见 *SQL and Relational Theory*）。

SPX，要么没有提供供应商 S2 所提供的零件，或者提供了供应商 S2 所提供的零件，那么就存在一个与供应关系 SPX 中零件号相等的供应关系 SPY。

9．查询零件重量大于 16 磅，或者由供应商 S2 提供的零件号码。

```
RANGEVAR PV RANGES OVER ( { PX.PNO } WHERE PX.WEIGHT > 16.0 ,
                          { SPX.PNO } WHERE SPX.SNO = 'S2' ) ;
{ PV.PNO } WHERE TRUE
```

强调几点：

- 该例中的域变量 PV 的取值范围是关系变量 P 和 SP 在 PNO 上的投影的并集。但要注意 PV 的定义要按照次序依赖于域变量 PX 和 SPX。
- 如果没有说明 WHERE 子句，则默认存在形如 WHERE TRUE 的子句。因而该例中的演算表达式就简化为：

```
{ PV.PNO }
```

10．查询每个零件的号码及重量，零件重量采用克表示。

```
{ PX.PNO , GMWT := PX.WEIGHT * 454 }
```

11．查询所有的供应商，每个供应商都赋予标识符“Supplier”作为标志。

```
{ SX , TAG := 'Supplier' }
```

12．查询每个供应关系的详细信息，包括供应零件的总量。

```
{ SPX , SHIPWT := PX.WEIGHT * SPX.QTY } WHERE PX.PNO = SPX.PNO
```

13．查询每个零件的号码及该零件的供应总数量。

```
{ PX.PNO , TQTY := SUM ( { SPX } WHERE SPX.PNO = PX.PNO , QTY ) }
```

调用 SUM 的第一个变量是一个关系，通过另一个关系演算表达式的形式进行说明。

14．查询供应商零件的总数量。

```
{ SUM ( { SPX } , QTY ) AS GRANDTOTAL }
```

15．查询存储量超过 5 的红色零件所在的城市。

```
{ PX.CITY } WHERE COUNT ( { PY } WHERE PY.CITY = PX.CITY
                                 AND PY.COLOR = 'Red' ) > 5
```

D.2 约束实例

这里给出了与第 6 章和第 13 章相同的 5 个样例（相同的 5 个商业规则）。

1．供应商的状态值必须在 1 至 100 之间。

```
CONSTRAINT CX1 NOT EXISTS SX ( SX.STATUS < 1 OR SX.STATUS > 100 ) ;
```

注意，如果检测约束条件时关系变量 S 恰好为空，则显然满足了约束条件。通常（不太严格）情况下，我们可以说当不存在任何 *X* 时（如域变量 *X* 的取值范围恰好为空集），量词表达式 EXISTS *X* (*bx*)返回值必须为 FALSE，无论布尔表达式 *bx* 取值如何。因此，在同样条件下，其否定形式 NOT EXISTS *X* (*bx*)的返回值就为 TRUE。

顺便说一下，依照上面的规则，如果不存在任何的 *X*，则 FORALL *X* (*bx*)的返回值也必须为 TRUE，同样无论布尔表达式 *bx* 取值如何。因此，该例的另一个存在争议的、但更具有用户友好性的格式如下：

```
CONSTRAINT CX1 FORALL SX ( SX.STATUS ≥ 1 AND SX.STATUS ≤ 100 ) ;
```

2．伦敦供应商的状态值必须为 20。

```
CONSTRAINT CX2 FORALL SX ( SX.CITY ≠'London' OR SX.STATUS = 20 ) ;
```

3．供应商号码必须是唯一的。

```
CONSTRAINT CX3 FORALL SX ( UNIQUE SY ( SX.SNO = SY.SNO ) ) ;
```

这个例子说明了量词的另一个作用（我们看到的第 3 种形式）。通常情况下，量词表达式 UNIQUE *X* (*bx*)的返回值为 TRUE，当且仅当恰好存在一个值（不包括超出域变量 *X* 范围的值的集合）使得布尔表达式 *bx* 的返回值为 TRUE。因此，该例表达式 FORALL SX (UNIQUE SY (SX.SNO = SY.SNO)) 的含义为“对于所有的供应商 SX，恰好存在一个与其具有相同号码的供应商 SY”。例如，如果 SX 表示供应商 S4 的 S 元组，则为了满足约束条件，SY 必须也表示供应商 S4 的 S 元组。换句话说，通常情况下，SX 和 SY 必须表示相同的元组。

4．状态值小于 20 的供应商不能供应零件 P6。

```
CONSTRAINT CX4
   FORALL SX ( SX.STATUS ≥ 20 OR
               NOT EXISTS PX ( PX.PNO = 'P6' AND
```

```
                                   EXISTS SPX ( SPX.SNO = SX.SNO AND
                                                SPX.PNO = PX.PNO ) ) ) ;
```

为了提起作者的兴趣，下面给出该约束条件的前束范式形式：

```
CONSTRAINT CX4
   FORALL SX ( FORALL SPX ( FORALL PX
        ( SX.STATUS ≥ 20 OR PX.PNO ≠ 'P6' OR
                            SX.SNO ≠ SPX.SNO OR SPX.PNO ≠ PX.PNO ) ) ) ;
```

5．SP 中的每个供应商号码都必须出现在 S 中。

```
CONSTRAINT CX5 FORALL SPX ( UNIQUE SX ( SX.SNO = SPX.SNO ) ) ;
```

D.3 简化语法

此部分给出了关系演算表达式的简化的 BNF 语法。注意，在实际语言中需要的许多特性（例如，书写关系标识符的能力）都故意被忽略了，是为了集中精力了解该语法的本质。

```
<range var def>
    ::= RANGEVAR <range var name> RANGES OVER <range> ;

<range>
    ::= <relvar name> | ( <relation exp commalist> )

<relation exp>
    ::= <proto tuple> [ WHERE <bool exp> ]

<proto tuple>
    ::= { <entry commalist> }

<entry>
    ::= <range var name> | [ <attribute name> := ] <attribute exp>

<attribute exp>
    ::= <exp>
```

注意：*<exp>*允许包含*<range var def>*，还可以使用恰当的类型标识符。

```
<range attribute ref>
    ::= <range var name> . <attribute ref>

<bool exp>
    ::=一切合法的表达式| <quantified bool exp>
```

注意：<*bool exp*>允许包含<*range var def*>，其中也可以使用恰当的类型标识符。

```
<quantified bool exp>
    ::= <quantifier> ( <bool exp> )

<quantifier>
    ::= <quantifier name> <range var name>

<quantifier name>
    ::= EXISTS | FORALL | UNIQUE
```

D.4 练习

写出下列查询和约束的关系演算表达式：

D.1 查询所有的供应关系。

D.2 查询由供应商 S2 提供的零件号。

D.3 查询状态值在 15～25 之间的供应商信息。

D.4 查询状态值小于供应商 S2 的状态值的供应商号码。

D.5 查询由巴黎所有供应商提供的零件号。

D.6 查询由伦敦供应商提供的零件号。

D.7 查询伦敦供应商没有提供的零件号。

D.8 查询所有的零件号码对，这些零件由某些供应商同时提供。

D.9 所有红色零件的重量必须小于 50 磅。

D.10 不能有两个供应商位于同一城市。

D.11 任何时候至多有一个供应商所在城市为雅典。

D.12 至少存在一个伦敦的供应商。

D.13 供应商 S1 和零件 P1 不能位于同一城市。

D.5 答案

在本附录中，假设域变量的形式为 SX、PX、SPX。

D.1 `{ SPX }`

D.2 `{ SPX.PNO } WHERE SPX.SNO = 'S2'`

D.3 `{ SX } WHERE SX.STATUS ≥ 15 AND SX.STATUS ≤ 25`

D.4 `{ SX.SNO } WHERE UNIQUE SY ( SY.SNO = 'S2' AND SX.STATUS < SY.STATUS )`

D.5 { SPX.PNO } WHERE FORALL SX (SX.CITY ≠ 'Paris' OR EXISTS SPY (SPY.SNO = SX.SNO AND SPY.PNO = SPX.PNO))

D.6 { SPX.PNO } WHERE UNIQUE SX (SX.SNO = SPX.SNO AND SX.CITY = 'London')

D.7 { SPX.PNO } WHERE FORALL SX (SX.SNO ≠ SPX.SNO OR SX.CITY ≠ 'London')

D.8 { XNO := SPX.PNO , YNO := SPY.PNO } WHERE SPX.SNO = SPY.SNO

D.9 CONSTRAINT DX9 FORALL PX (PX.WEIGHT < 50) ;

D.10 CONSTRAINT DX10 FORALL SX (UNIQUE SY (SX.CITY = SY.CITY)) ;

D.11 CONSTRAINT DX11 COUNT ({ SX } WHERE CITY = 'Athens') < 2 ;

D.12 CONSTRAINT DX12 EXISTS SX (SX.CITY = 'London') ;

D.13 CONSTRAINT DX13 NOT EXISTS SX (NOT EXISTS PX (SX.SNO = 'S1' AND PX.PNO = 'P1' AND SX.CITY ≠ PX.CITY)) ;

附录 E

进阶阅读指南

我不介意你慢慢思考，
但我介意你的出版物比你想象的要快。
——*Wolfgang Pauli*（节选）

关系数据库技术含有丰富的内容，其中一项就是与 SQL 有关的。这个附录从你可能感兴趣的两个来源对已经发表的观点进行了解释（以专著或者论文的形式）。注意：首先我要致歉，因为我的名字多次以作者或联合作者的方式出现在列表中，而不管材料的特点如何，这种状态或多或少是不可避免的（如果你原谅我这样说的话）。

E.1 E.F.Codd 的论文

- “Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks”，源自 IBM 研究报告 RJ599（1969 年 8 月 19 日发表）。“A Relational Model of Data for Large Shared Data Banks”，源自 *CACM13* 第 6 期（1970 年 6 月）。注意：第一篇论文后来又重新发表在 *ACM SIGMOD Record 38* 第 1 期（2009 年 3 月），第二篇论文重新发表在 CACM26 第 1 期 *Milestones of Research* 上，选择的都是 1958 年～1982 年的论文（*CACM 25th Anniversary Issue*）。

  1969 年发表的论文是 Codd 第一次阐述关系模型，是 1970 年发表的论文的基础。但 1970 年发表的论文在某些兴趣领域有所差异，主要是 1969 年的论文中允许使用关系的值属性，但 1970 年发表的论文中没有使用。因此在 1970 年的论文中，Codd 第一次拓宽了应用领域，1970 年的论文被认为是该领域中的一个标志性的论文，但这么说对于 1969 年的先驱者来说有些不公平，所以我建议每一位数据库专业人士每年至少阅读一篇这样的论文。

- “Relational Completeness of Data Base Sublanguages”，作者为 Randall J. Rustin，发表在 *Data Base Systems*, *Courant Computer Science Symposia Series 6*，Englewood Cliffs, NJ:Prentice Hall（1972）。
  在这篇论文中，Codd 第一次整理出了最初的关系代数和关系演算的定义。虽然阅读起来不太容易，但这是对缜密科学研究的回报。
- “Further Normalization of the Data Base Relational Model”，作者为 Randall J. Rustin，发表在 *Data Base Systems, Courant Computer Science Symposia Series 6*，Englewood Cliffs, NJ:Prentice Hall（1972）。
  在该论文中，Codd 定义了 2NF 和 3NF，为数据库设计理论奠定了领域研究基础。但标题容易误导读者，进一步规范化不是在关系模型上完成的事情，而是对关系变量的进一步处理或者是对关系变量设计的进一步处理。（在第 9 章我们已经看到，关系模型本身并不关心数据库是如何设计的，而只是关心问题的设计是否破坏了关系模型的规则。）警告：该论文中的 1NF 定义与本书前面给出的定义有所区别，实际上是不正确的（我将给出理由证明）。

E.2 C. J. Date 的著作

- *An Introduction to Database Systems*（第 8 版）， Boston, MA: Addison- Wesley（2004）。
  面向大学层次的论著，讨论了数据库管理的所有方面（不单是关系型的）。SQL 语言以 SQL:1999 为范例，但对 SQL:2003 给出了一些注释。而且还专门对 SQL 的“对象/关系型”特征进行了详细讨论（REF 类型、引用值等），并详细解释了它们破坏关系型规则的原因。注意，还有一些涉及这个主题的论著如下：
  - Raghu Ramakrishnan 和 Johannes Gehrke：*Database Management Systems* (3rd edition). New York, NY: McGraw-Hill（2003）。
  - Ramez Elmasri 和 Shamkant Navathe：Fundamentals of Database Systems (6th edition). Boston, MA: Addison-Wesley（2010）。
  - Avi Silberschatz, Henry F. Korth 和 S. Sudarshan：Database System Concepts (6th edition). New York, NY: McGraw-Hill（2010）。
  - *Go Faster*！*The TransRelationalTM Approach to DBMS Implementation*. Frederiksberg，Denmark：Ventus（2002，2011）。可以从 www.bookboon.com. 上免费下载。
    该论著描述了一种全新的数据库管理系统的实现方法，它与原来各种基

本的实现方法有明显的区别，这些基本的实现方法在当前使用的主流SQL 产品中都可以看到。

- *Date on Database*：*Writings* 2000-2006，Berkeley，CA: Apress（2006）。
该论著是有关数据库各种话题的文章集合，有几篇是直接与此书讨论的主题相关的。特别是它详细讨论了 SQL 与关系模型的不同点，以及因此而产生的一些负面影响。相关的章节是：第 8 章“What First Normal Form Really Means”、第 9 章“A Sweet Disorder”（关于从左到右的列排序问题）、第 10 章“Double Trouble, Double Trouble”（重复值引起的问题）、第 18 章“Why Three-and Four-Valued Logic Don’t Work”（支撑 SQL 的空值定义而引起的逻辑假设问题）。
- *SQL and Relational Theory: How to Write Accurate SQL Code* (2nd edition). Cambridge，MA: O’Reilly（2012）。
该论著在本书前面反复被引用，题目简写为 *SQL and Relational Theory*。该论著中讨论的主题以及在本书中没有详细讨论的内容包括：视图和其他衍生的关系变量、重复值的进一步讨论、类型理论、关系值属性、多重赋值、SQL 语法、递归查询、元组等价的重要性、选择器和 THE 运算符、空值和 3 值逻辑、如何处理丢失的信息、数据维度、优化和表达式转换、域值检测、谓词逻辑、更多的关系运算符、关系常量及其他。更多的背景知识可以参见本书前言中介绍的内容。
- *Database Design and Relational Theory: Normal Forms and All That Jazz.* Cambridge，MA: O’Reilly（2012）。
- *View Updating and Relational Theory: Solving the View Update Problem. Cambridge*，MA: O’Reilly（2013）。

E.3 C. J. Date and Hugh Darwen 的著作

- *A Guide to the SQL Standard* (4th edition)，Boston，MA: Addison-Wesley（1997）。
该论著主要介绍了 SQL:1992，但也包含了调用级接口特征 CLI（1995 年发布）、永久性存储模块特征 PSM（1997 年发布），以及后来成为 SQL:1999 一部分的一些特征。尽管这个标准在 2011 年被重新编辑了，但不公正地讲，该著作包含了对 SQL 关系型特征可靠的引用说明和使用指南。
- *Databases, Types, and the Relational Model*: *The Third Manifesto*（3rd edition），Boston，MA: Addison-Wesley（2007）。
该论著介绍并解释了第三次发布的数据库、类型及关系模型的相关细节，

给出了一个准确的、很正式的关系模型定义，以及支持类型的理论（包括类型继承的综合模型）。注意，这个宣言本身只是由该论著中很短一个章节组成（12 页，整个论著超过了 500 页 。

- *Database Explorations: Essays on The Third Manifesto and Related Topics*，Bloomington，IN: Trafford（2010）。

  此外，该论著包含了第三次宣言以及 Tutorial D 语言的当前修订版本。（请注意，过去 Tutorial D 并不是第三次宣言本身的一部分，只是用来解释这个宣言核心思想的一种语言。）该论著也包含了将 Tutorial D 推广到工业级应用的一些假设。注意：网站 www.thethirdmanifesto.com 给出了 Tutorial D 实现的各种细节信息，以及关于此宣言的其他细节，可以参见网页 http://dbappbuilder.sourceforge.net/rel.html，其中提供了下载 Rel（由 Dave Voorhis 提供的 Tutorial D 原型实现）代码的接口。

顺便说一下，该论著第 27 章的附录“Is SQL's Three-Valued Logic Truth Functionally Complete?”包含了对 SQL 空值及与空值有关的特征的详细的、综合的、缜密的描述。

E.4 与 SQL 相关的其他出版物

- Donald D. Chamberlin 和 Raymond F. Boyce：“SEQUEL: A Structured English Query Language”，1974 年 ACM SIGMOD 工作组发布，主要内容描述了数据描述、访问和控制，1974 年 5 月，Ann Arbor, MI 出版。

  就像第 10 章给出的注释一样，这是第 1 篇介绍 SQL 语言的论文（最初的名字是 SEQUL，即结构化英文查询语言）。该论文中描述的 SEQUL 与现在通常意义上理解的 SQL 存在一些有意思的差异，例如：

  - 没有空值。
  - 虽然支持 SELECT 子句，但不存在“SELECT *”，因此如果想获得伦敦所有的供应商，就必须表示为：S WHERE CITY = ‘London’。为了获得所有的供应商，必须表示成 S。
  - 默认消除了重复值（没有在“集合函数”中使用，因为这是很不恰当的，只是在 SQL 标准中作为术语使用，用来引用 SUM 和 AVG）。
  - FROM 子句中只能包含一个表名。换句话说，SEQUL 不能实现联接运算。
  - WHERE 子句中比较式右边的比较数可以是一个子查询（虽然不存在子查询这个术语，但在表达式中用映射替代），这种情况下的比较式通常是 ANY 或 ALL 的比较（参见 *SQL and Relational Theory*）。ANY 是默

认的，可以隐式说明，但语法本身不支持，使用“=”替代（默认为“=ANY”）。

- 支持集合比较运算符（如：集合包含等）。
- 没有 GROUP BY 子句，GROUP BY 功能的确存在，但需要在 FROM 子句中说明。
- 没有 HAVING 子句，但在 WHERE 子句中可以调用“集合函数”来替代此功能。
- 没有“关联名”（这是最初的 SEQUL 术语中的 SQL 的域变量）。相反，可以标识一些块（在此环境下，它是映射的另一种说法），块标签可以通过（.）量词来引用。
- 在 SELECT 子句中诸如 QTY 或 AVG（QTY）的表达式是合法的（即表达式中涉及“集合函数”调用，或者对列的引用等等），这类表达式也可以用在 WHERE 子句中。
- “映射”可以进行交运算、并运算、差运算等（这些运算符都采用常用的数学符号表示，而不是关键词）。

该论文还讨论了 SEQUL 与关系演算之间的一些差异，声明在任何情况下 SEQUEL 都优于关系演算。然而，这些声明和差异都不能经受住仔细的推敲和分析。

- 国际标准化组织（ISO）：*Database Language SQL*，Document ISO/IEC 9075:2011（2011）。

  它是官方发布的 SQL 标准（2011 版）。注意它实际上是国际标准，不只是美国国家或“ANSI”标准（很多人曾这样认为过）。还要注意到，虽然 SQL:2011 是目前使用的标准版本，但基本上此书中讨论的所有 SQL 特征都已经包含在了 SQL:1992 或者更早的版本中。
- Jim Melton 和 Alan R. Simon：“SQL:1999—Understanding Relational Components”、Jim Melton：“SQL:1999—Understanding Object-Relational and Other Advanced Features”，分别于 2002 年和 2003 年由 San Francisco、CA: Morgan Kaufmann 发表。

  迄今为止，我知道这两部是仅有的、可以使用的、内容覆盖度广的两部。它们之中的任何一个标准都晚于 SQL:1992。Melton 担任该标准的编辑很多年了（现在仍然是，而且是在写作的同时来担任此项工作。）
- Stéphane Faroult 和 Peter Robson：*The Art of SQL*. Cambridge, MA: O’Reilly（2006）。

  该著作是实践者的指南，可以知道你如何在当前的产品中更好地发挥 SQL 的作用。下面是经过编辑的该著作的 12 个章节标题：

1．设计高性能的数据库
2．高效地访问数据库
3．索引
4．理解 SQL 的语句
5．理解物理层实现
6．典型的 SQL 特性
7．处理异构数据
8．复杂案例
9．并发
10．大数据
11．响应时间
12．性能监测

该著作在它的意见和建议中没有偏离太多的关系原理，实际上大部分章节中还显示地表明了对这些原理的支持。但是它也承认当今的优化器还不是很完美，因此如何针对特定的问题在许多逻辑上等价的格式之外来选择特定的 SQL 格式，该著作中给出了一些建议，可能会使性能变得更好（而且还揭示了具体原因）。

E.5 其他图书

- Patrick Hall、Peter Hitchcock 和 Stephen Todd：“An Algebra of Relations for Machine Computation”，Conf. Record of the 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, CA（January 1975）。
  该论文也许有点“难”，但我想它还是很重要的。在本书中我曾经描述过的 Tutorial D 和关系代数的版本都可以在这篇论文中找到根源。
- Jim Gray and Andreas Reuter：*Transaction Processing*: *Concepts and Techniques*，San Francisco，CA: Morgan Kaufmann（1993）。
  与事务管理相关的标准文档（见本书第 8 章 。
- Lex de Haan 和 Toon Koppelaars：*Applied Mathematics for Database Professionals*，Berkeley，CA: Apress（2007）。
  除了上面的内容以外，该著作还以过程代码的方式介绍了实现完整性约束的实现过程（如果需要，请参见本书第 13 章 。

特 别 优 惠

购买本书的读者专享**异步社区购书优惠券**。

使用方法：注册成为社区用户，在下单购书时输入 57AWG 使用优惠码，然后点击“使用优惠码”，即可享受电子书 8 折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：


异步社区


微信服务号


微信订阅号


官方微博


QQ 群：436746675

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@ 人邮异步社区，@ 人民邮电出版社 - 信息技术分社

投稿 & 咨询：contact@epubit.com.cn